TURING 图灵程序设计丛书

PACKT PUBLISHING



[印度] Ved Antani 著 门佳 译

JavaScript编程精粹

Mastering JavaScript

掌握JavaScript基础知识要点及其现代技术和工具，
用正确的编码风格开发Web应用

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

作者简介

Ved Antani

印度知名时尚电商Myntra的副总裁助理，曾就职于Electronic Arts、NetApp和Oracle。从2005年起一直在从事JavaScript、Go和Java编程，在构建可伸缩系统和移动应用开发方面有着丰富的经验。

译者简介

门佳

资深GNU/Linux用户，喜欢溯本求源，挖掘技术背后的来龙去脉。对于程序语言设计理论、编译技术、操作系统设计与实现、Web开发等领域均有涉猎，译著包括《Linux Shell脚本攻略》《TCP Sockets编程》《精通JavaScript（第2版）》《Linux命令行与shell脚本编程大全（第3版）》等。

图灵程序设计丛书

JavaScript编程精粹

Mastering JavaScript

[印度] Ved Antani 著　门佳 译

人 民 邮 电 出 版 社
北　京

图书在版编目（C I P）数据

JavaScript编程精粹 / (印) 韦德·安塔尼
(Ved Antani) 著 ; 门佳译. -- 北京 : 人民邮电出版社,
2017.8
(图灵程序设计丛书)
ISBN 978-7-115-46569-6

Ⅰ. ①J… Ⅱ. ①韦… ②门… Ⅲ. ①JAVA语言—程序
设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第181430号

内 容 提 要

本书详细介绍讲述了 JavaScript 的基础知识以及一些现代语言工具和库，例如 jQuery、Underscore.js 和 Jasmine。主要内容包括：JavaScript 基础知识，函数、闭包和模块，数据结构和相关处理，面向对象的 JavaScript，JavaScript 设计模式，测试与调试，ECMAScript 6，DOM 事件和操作，服务器端 JavaScript。

本书适合所有 JavaScript 开发人员阅读。

◆ 著　　[印度] Ved Antani
译　　门　佳
责任编辑　岳新欣
执行编辑　赵雪梅
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：11.5
字数：272千字　　　　2017年 8 月第 1 版
印数：1 – 3 500册　　　　2017年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-6690号

定价：39.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

致　谢

感谢妻子Meghna对我的支持。她总会在我最需要的时候给予我鼓励和帮助。

前　　言

看起来，与JavaScript有关的内容该写的都已经写过了。坦白说，很难找出JavaScript中还有什么没被人翻来覆去讨论过的话题。但是，JavaScript的发展速度太快了。ECMAScript 6有可能会给这门语言以及编码方式带来巨大的转变。Node.js已经改变了我们使用JavaScript编写服务器的方法，而像React和Flux这些新理念将会推动JavaScript的再次迭代。我们在学习这些新特性的同时，不应该忽略那些必须掌握的JavaScript基础知识。这些知识是根基，不容忽视。如果你已经是一位JavaScript的开发老手，会意识到现代JavaScript已经与大多数人记忆中的它大相径庭了。现代JavaScript不但要求遵循特定的编码风格以及严格的设计思路，而且编程工具的功能愈发强大，已经逐渐成为了开发流程中不可或缺的组成部分。尽管JavaScript在不断改变，但它是根植于一些非常坚实、稳定的概念之上的。本书强调的正是这些基本概念。

在本书写作之时，JavaScript的变化仍未停歇。幸运的是，所有重要的相关更新都已经包括在了本书中。

本书为你详细讲述了JavaScript的基础知识，以及一些现代的语言工具和库，例如jQuery、Underscore.js和Jasmine。

希望你能够享受阅读本书的过程，就像我们享受写作本书的过程一样。

内容简介

第1章，JavaScript入门。本章的重点在于语言构件（language construct），在基本细节方面不会花费过多篇幅。本章将讲述变量作用域和循环中不太容易掌握的地方，以及类型和数据结构的最佳使用实践。另外，还包括一些编码风格的相关知识，以及推荐的代码组织模式。

第2章，函数、闭包与模块。本章将讲述JavaScript语言错综复杂的核心，探讨在JavaScript中，因闭包的不同用法而造成的复杂性。本章细致详尽的讨论将为你今后深入学习更高级的设计模式奠定基础。

第3章，数据结构及相关操作。本章将详细讲述正则表达式和数组。数组是JavaScript的基本数据类型，本章将帮助你学会有效地使用数组。正则表达式能够使代码更简洁，我们将仔细讲解

如何充分发挥出正则表达式的功能。

第4章，面向对象的JavaScript。本章将讲述JavaScript中的面向对象编程，包括继承和原型，重点将放在理解JavaScript的原型继承模型上。另外，还将讨论该模型与其他面向对象模型之间的差异，以帮助Java或C++程序员熟悉这种变化。

第5章，JavaScript模式。本章将论述常见的设计模式以及如何在JavaScript中实现这些模式。一旦你掌握了JavaScript的面向对象模型，就更容易理解设计和编程模式，从而写出易于维护的模块化代码。

第6章，测试与调试。本章将讲述各种现代化的测试方法以及JavaScript的调试问题。另外，还将探究JavaScript的持续测试和测试驱动方法。本章中将采用Jasmine作为测试框架。

第7章，ECMAScript 6。本章将重点放在ECMAScript 6（ES6）所引入的新语言特性上。它们使得JavaScript的功能更加强大，本章将帮助你理解并使用这些新特性。

第8章，DOM操作与事件。本章将JavaScript作为一种浏览器语言进行了详尽的描述，讨论了DOM操作以及浏览器事件。

第9章，服务器端JavaScript。本章将讲解如何使用Node.js编写可伸缩的服务器系统，探讨Node.js的架构和一些实用技术。

阅读前提

本书中所有的示例都可以运行在任何现代浏览器中。学习最后一章时，你需要安装Node.js。运行书中的示例需要满足以下前提条件。

- 一台安装了Windows 7（或更高版本）、Linux或Mac OS X操作系统的计算机。
- 最新版的Google Chrome或Mozilla Firefox浏览器。
- 选择一款文本编辑器。Sublime Text、vi、Atom或者Notepad++都挺不错。选择权完全在你。

目标读者

本书旨在教授精通JavaScript必须掌握的基础知识，适合以下读者。

- 具备其他面向对象语言经验的开发人员。书中的内容能够帮助他们利用已有的经验转换到JavaScript。
- 比较熟悉JavaScript的Web开发人员。本书将帮助他们学习JavaScript的高级概念，改善他们的编程风格。

- ❑ 希望了解并最终成为JavaScript高手的初学者。本书包含了必不可少的入门内容。

排版约定

在本书中，你会发现针对不同信息类型的文本样式。下面是这些样式的示例和解释。

正文中的代码、数据库表名、用户输入的样式如下："第一种方式是通过`<head>`中的`<script>`标签导入JavaScript，第二种方式是利用`<script>`标签嵌入内联JavaScript。"

代码块的版式如下：

```
function sayHello(what) {
  return "Hello " + what;
}
console.log(sayHello("world"));
```

当需要读者特别注意代码块中的某一部分时，相关的代码行和项将以粗体显示：

```
<head>
  <script type="text/javascript" src="script.js"></script>
  <script type="text/javascript">
    var x = "Hello World";
    console.log(x);
  </script>
</head>
```

命令行输入或输出形式如下：

```
EN-VedA:~$ node
> 0.1+0.2
0.30000000000000004
> (0.1+0.2)===0.3
false
```

新术语或**重要词汇**会以黑体显示。



此图标表示警告或重要的注意事项。



此图标表示提示和技巧。

读者反馈

欢迎读者反馈意见。我们想知道读者对于本书的看法——喜欢哪些内容或不喜欢哪些内容。读者反馈对于我们出版读者真正需要的图书至关重要。

如有反馈意见，请将电子邮件发送到feedback@packtpub.com，不要忘记在邮件标题中注明你要反馈的书名。

客户支持

现在你已经成为Packt图书的主人了，为了使此书尽可能物有所值，我们还提供了其他服务。

下载示例代码

你可以用你的账户从http://www.packtpub.com中下载所购买的所有Packt图书的示例代码。如果你是从其他地方购买的本书（英文版），可以访问http://www.packtpub.com/support并注册，以便通过电子邮件取得示例代码。

下载彩色图片

我们为你提供了一份PDF文件，其中包含了书中出现的所有截屏和图示的彩色图片。这些彩色图片有助于你更好地理解输出内容的变化。你可以从这里下载到该文件：https://www.packtpub.com/sites/default/files/downloads/MasteringJavaScript_ColorImages.pdf。

勘误

尽管我们已竭尽全力确保本书内容的准确性，但错误终难避免。如果你发现了书中的错误（不管是文字错误还是代码错误）并愿意告知我们，我们将非常感激。这样不仅可以减少其他读者的疑惑，也有助于本书后续版本的改进。要提交所发现的错误，请访问http://www.packtpub.com/submit-errata，选择书名，点击Errata Submission Form（勘误提交表单）链接，输入详细的错误信息。[①]一旦勘误得到核实，我们将接受你的提交，同时勘误内容也会被上传到我们的网站，或是添加到对应书目勘误区的现有勘误表中。

要想查看之前提交的勘误，进入https://www.packtpub.com/books/content/support，在搜索框中

① 本书中文版勘误可到www.ituring.com.cn/book/2069查看和提交。——编者注

输入书名即可。你所需的信息会出现在Errata（勘误）下方。

举报盗版

各类媒体在网络上一直饱受版权侵害的困扰。Packt坚持不懈地严格保护版权和授权。如果你在网上发现了我社图书的任何形式的盗版，请立即为我们提供地址或网站名称，以便我们采取措施。

请将疑似侵权的网站链接发送至copyright@packpub.com。

衷心感谢你为保护作者的知识产权以及我们的劳动成果所做的工作。

疑难解答

如果你对本书的任何方面有疑问，请通过questions@packtpub.com联系我们，我们将尽力为你解决。

电子书

扫描如下二维码，即可购买本书电子版。

目　　录

第 1 章 JavaScript入门

万事开篇难，尤其是对于像JavaScript这种主题。为难之处主要在于这门语言已经被讨论过太多了。JavaScript已经成为Web世界的语言——在Netscape Navigator出现之初，它就作为一种通用语言存在了。从业余爱好者的一件工具摇身变为行家手中的利器，JavaScript发展之快令人瞠目结舌。

JavaScript是Web上最流行的语言和开源生态环境。http://githut.info/跟踪记录了这门语言过去几年中在GitHub上活跃仓库的数量以及整体受欢迎程度。JavaScript如此流行且重要，归功于它同浏览器的紧密结合。作为经过极度优化的JavaScript引擎，Google的V8和Mozilla的SpiderMonkey分别为Google Chrome和Mozilla Firefox浏览器的壮大贡献了一臂之力。

尽管Web浏览器是JavaScript应用最广泛的平台，但如今像MongoDB和CouchDB这类现代数据库，也开始使用JavaScript作为其脚本和查询语言。JavaScript已经成为浏览器之外的一个重要平台。例如，Node.js和io.js项目就为使用JavaScript开发可伸缩的服务器环境提供了强有力的平台；另外一些引人注目的项目更是将这门语言的性能推向了极限，例如Emscripten（http://kripken.github.io/emscripten-site/），这是一个基于**低层虚拟机**（Low-Level Virtual Machine，LLVM）的项目，它能够将C和C++代码编译成高度优化的asm.js格式的JavaScript代码。这使得你可以在Web上以接近于原生速度运行C/C++代码。

JavaScript有着坚实的构建基础，其中包括函数、动态对象、松散类型、原型继承以及强大的对象字面记法。

尽管JavaScript的设计理念非常不错，但遗憾的是，它只能跟随浏览器的发展。在支持各种特性和标准方面，Web浏览器声名狼藉。JavaScript试图去适应浏览器中各种奇思妙想的设计，结果作出了一些非常糟糕的设计决策。多数人被这些不良成分（bad parts，借由Douglas Crockford火起来的一个术语）蒙蔽了双眼，忽视了语言中那些优秀的部分。一些程序员写出的差劲代码成为调试这些代码的另一批程序员的噩梦，而JavaScript则背负了骂名。因此，JavaScript不幸地成为被误解最多的编程语言之一（http://javascript.crockford.com/javascript.html）。

对JavaScript的另一种批评是，你不用真正掌握这门语言就可以把问题搞定。我见过一些程序

员写出了糟糕得令人发指的JavaScript代码，仅仅就是因为他们想尽快完成任务，而JavaScript偏偏也允许他们这么干。我曾花费大量时间来调试一个压根就算不上程序员的家伙编写的JavaScript代码，质量真是差劲到家了。但是你不能因为自己的编程水平不到家，就把怨气撒在语言身上，毕竟它只是一种工具而已。就像所有的手艺一样，编程也需要全身心的投入和自律。

1.1　JavaScript 极简史

1993年，美国国家超级计算机应用中心（NCSA）的Mosaic浏览器是当时流行的首批Web浏览器之一。一年后，网景公司推出了其专有的Web浏览器：Netscape Navigator。多名Mosaic的早期设计人员参与了Navigator的开发。

1995年，网景公司雇用了Brendan Eich，允诺他在浏览器中实现Scheme（一种Lisp方言）。在此之前，网景公司与Sun公司（已被Oracle公司收购）进行过接触，试图将Java纳入到Navigator浏览器中。

由于Java的流行性和易用性，网景公司决定自家浏览器中的脚本语言的语法必须和Java类似。这直接就把已有的那些语言，如Python、TCL（Tool Command Language，工具命令语言）或Scheme排除在外了。1995年5月，Eich只花了10天时间就写出了最初的原型（http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf）。 JavaScript一开始的代号是Mocha，是由Marc Andreessen命名的。由于注册商标方面的原因，网景公司随后将其更名为LiveScript。1995年12月初，Sun公司将注册商标Java授权给了网景公司，于是这门语言终于拥有了最终的名字：JavaScript。

1.2　如何阅读本书

这并不是一本快餐类读物。本书关注的是正确的JavaScript编程方式。我们会用大量的篇幅来学习如何避开该门语言中的不良成分，构建可靠、可读的JavaScript代码。为了避免对那些糟糕的语言特性形成依赖，我们选择对其敬而远之；如果已经养成了不好的编码习惯，本书将尽力帮你摆脱这种问题。本书会重点强调使用正确的风格和工具来改善代码。

本书中的大部分概念都反映了现实世界中的问题和模式。我坚持认为读者应该自己动手输入每一段代码，以确保彻底理解这些概念。相信我，要学编程，没有什么方法比动手编写大量代码更好的了。

通常需要创建一个HTML页面来运行嵌入的JavaScript代码：

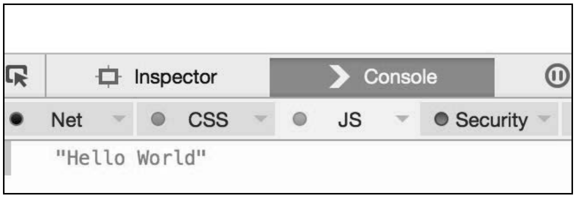
```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="script.js"></script>
```

```
  <script type="text/javascript">
    var x = "Hello World";
    console.log(x);
  </script>
</head>
<body>
</body>
</html>
```

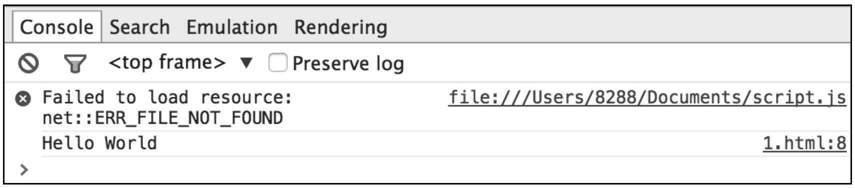
这段示例代码展示了在HTML页面嵌入JavaScript代码的两种方式。第一种方式是通过`<head>`中的`<script>`标签导入JavaScript，第二种方式是利用`<script>`标签嵌入内联JavaScript。

> **下载示例代码**
>
> 可以用你的账户从http://www.packtpub.com中下载所购买的所有Packt图书的示例代码。如果你是从其他地方买的（英文版），可以访问http://www.packtpub.com/support并注册，通过电子邮件取得示例代码。

可以把HTML页面保存在本地，然后在浏览器中打开。在FireFox中，打开Developer控制台（FireFox菜单 | Developer | Web Console），此时会看到在Console标签下出现了`"Hello World"`文本。在你用的操作系统和浏览器上，屏幕显示可能会不太一样：



打开页面，然后使用Chrome的Developer Tool来进行检查：



这里我们注意到一个很有意思的现象：在控制台中出现了一个丢失.js文件的错误信息，这个文件是通过下面这行代码导入的：

```
<script type="text/javascript" src="script.js"></script>
```

使用浏览器的开发者控制台或是扩展（如Firebug），可以非常方便地调试代码中出现的错误。在随后的章节中，将详细讨论相关的调试技巧。

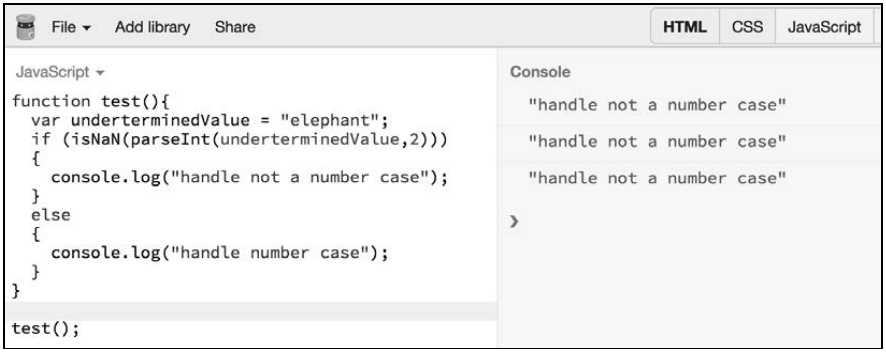
为书中的每一个练习创建这种HTML脚手架（scaffold）很是乏味，因此我们打算使用一种用于JavaScript的**读取–求值–输出–循环**（Read-Eval-Print-Loop，REPL）。JavaScript不像Python那样配备了REPL，我们可以把Node.js作为替代。如果已经安装了Node.js，那么只需要在命令行中输入`node`就可以开始动手练习了。你会注意到，Node REPL显示的错误信息实在是不怎么像样。

来看看下面的例子：

```
EN-VedA:~$ node
>function greeter(){
  x="World"l
SyntaxError: Unexpected identifier
  at Object.exports.createScript (vm.js:44:10)
  at REPLServer.defaultEval (repl.js:117:23)
  at bound (domain.js:254:14)
  ...
```

出现错误之后，只能从头再来。不过，它仍然可以帮助你快速测试短的代码片段。

我个人经常使用的另一个工具是JS Bin（http://jsbin.com/）。JS Bin提供了大量的JavaScript测试工具，例如语法高亮和运行时刻错误检测。下面是JS Bin的运行界面：



可以根据自己的喜好，选择能够简化示例代码测试的工具。不管选择哪种工具，一定不要错过书中任何一个练习。

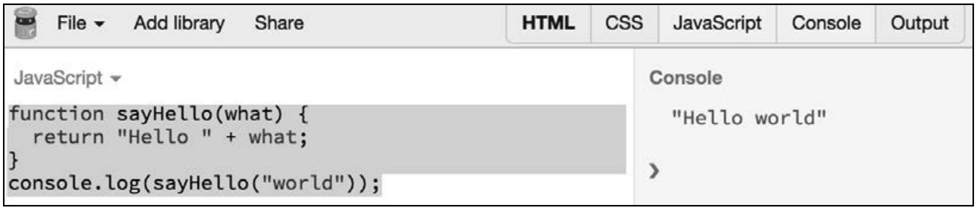
1.3　Hello World

任何一门编程语言都不能没有Hello World程序——本书怎么可能例外呢？

在JS Bin中输入以下代码（不要复制粘贴）：

```
function sayHello(what) {
  return "Hello " + what;
}
console.log(sayHello("world"));
```

屏幕显示如下内容：



1.3.1 JavaScript 概览

简而言之，JavaScript是一种基于原型的脚本语言，支持动态类型和头等函数。JavaScript的大部分语法借鉴了Java，但同时也受到了Awk、Perl和Python的影响。JavaScript区分大小写，并且不会受到空白字符的影响。

1. 注释

JavaScript允许单行或多行注释，其语法类似于C或Java：

```
// 单行注释

/* 这是一个更长的、
   占据了多行的注释
*/

/* 你不能 /* 嵌套注释 */ SyntaxError */
```

2. 变量

变量是值的符号名称。变量名，或者说是标识符，必须遵循特定的规则。

JavaScript的变量名必须以字母、下划线（_）或是美元符（$）开头，随后的字符可以包含数字（0–9）。由于JavaScript区分大小写，因此字母也就包括字符A–Z（大写）以及字符a–z（小写）。

也可以在变量名中使用ISO 8859-1或者Unicode中的字母。

JavaScript中的新变量应该使用关键字var定义。如果声明了一个变量，但是没有给它赋值，那么该变量的类型默认是未定义的。有一件很糟糕的事情，如果没有使用关键字var声明变量，这种变量会成为隐式全局变量（implict global）。再重申一遍：隐式全局变量可不是什么好东西。等讲到变量作用域和闭包的时候，我们会详细地讨论这个话题。重要的是要记住：除非知道自己在做什么，否则要坚持使用var关键字来声明变量：

```
var a;          //声明一个未定义的变量
var b = 0;
console.log(b);     //0
```

```
console.log(a);      //undefined
console.log(a+b);    //NaN
```

NaN是一个特殊的值，表明内容不是数字。

3. 常量

可以使用const关键字创建只读的命名常量。常量名必须以字母、下划线或美元符开头，余下部分可以包含字母、数字或下划线字符：

```
const area_code = '515';
```

不能通过赋值或重新声明来修改常量的值，常量必须被初始化成一个值。

JavaScript支持以下标准类型：

- Number（数值）
- String（字符串）
- Boolean（布尔）
- Symbol（ECMAScript 6中的新类型，符号）
- Object（对象）
  - Function（函数）
  - Array（数组）
  - Date（日期）
  - RegExp（正则表达式）
- Null（空）
- Undefined（未定义）

4. Number

Number类型能够描述32位整数和64位浮点数的值。例如，下面的代码声明了一个能够保存整数值的变量，其中的整数值是通过字面量555来定义的：

```
var aNumber = 555;
```

要定义浮点数值，需要加入一个小数点和小数点之后的一个数字：

```
var aFloat = 555.0;
```

其实，JavaScript中并不存在整数。JavaScript是使用64位浮点数来描述整数的，和Java采用双精度浮点数的做法一样。

因此，你会看到如下结果：

```
EN-VedA:~$ node
> 0.1+0.2
0.30000000000000004
> (0.1+0.2)===0.3
false
```

推荐你去StackOverflow（http://stackoverflow.com/questions/588004/is-floating-point-math-broken）和http://floating-point-gui.de/阅读有关该问题的详尽解答。重要的是要明白涉及浮点数运算的时候应该加倍小心。在大多数情况下，并不是非得依赖高精度小数，但如果真的需要的话，可以尝试使用big.js（https://github.com/MikeMcl/big.js）这类库来解决问题。

如果编码的对象是有极高精度要求的金融系统，应该以分为单位来描述金额，以避免取整错误。我曾经工作过的一个系统会将增值税（VAT）取整到两位小数。如果每份订单都按照这种方法取整，一天数千份的订单会让会计头疼死。为此，我们将整个Java Web服务栈和JavaScript前端进行了一次彻底整修。

有几个特殊值也是Number类型的一部分。头两个是`Number.MAX_VALUE`和`Number.MIN_VALUE`，它们定义了Number类型值的上下边界。所有的ECMAScript数字一定都处于这个区间内，概莫能外。但是运算能够产生一个不在此区间的数字。当运算产生的数字大于`Number.MAX_VALUE`时，运算结果被赋以`Number.POSITIVE_INFINITY`，表示该结果不再是数字值。与此类似，如果运算产生的数字小于`Number.MIN_VALUE`，运算结果被赋以`Number.NEGATIVE_INFINITY`，同样表示非数字值。如果运算返回的值是无穷，则该结果不能用于后续的运算。可以使用`isInfinite()`方法来验证运算结果是否为无穷。

JavaScript另一个特殊之处在于它有一个叫作NaN（Not a Number的缩写）的特殊值。一般来说，这个值会出现在类型（String、Boolean等）转换失败的时候。观察下面代码中NaN的特点：

```
EN-VedA:~ $ node
> isNaN(NaN);
true
> NaN==NaN;
false
> isNaN("elephant");
true
> NaN+5;
NaN
```

第二行很奇怪——NaN不等于NaN。如果NaN出现在数学运算中的任何一部分，结果都会变成NaN。作为一个通用规则，不要在任何表达式中使用NaN。对于高级数学运算，可以使用`Math`全局对象及其方法：

```
> Math.E
2.718281828459045
> Math.SQRT2
1.4142135623730951
```

```
> Math.abs(-900)
900
> Math.pow(2,3)
8
```

可以使用parseInt()和parseFloat()方法将字符串表达式转换成整数或浮点数：

```
> parseInt("230",10);
230
> parseInt("010",10);
10
> parseInt("010",8); //八进制
8
>parseInt("010",2); //二进制
2
>+ "4"
4
```

使用parseInt()的时候，应该明确地给出一个基数，避免在老版本的浏览器上出现出人意料的结果。最后一个技巧是使用+号将字符串"42"自动转换成数字42。使用isNaN()处理parseInt()的结果也是一种明智的做法。让我们来看看下面的例子：

```
var underterminedValue = "elephant";
if (isNaN(parseInt(underterminedValue,2)))
{
   console.log("handle not a number case");
}
else
{
   console.log("handle number case");
}
```

在这个例子中，如果变量underterminedValue的值是通过外部接口设置的，则你无法确定该变量值的类型。如果不使用isNaN()进行处理，parseInt()将会产生异常，导致程序崩溃。

5. 字符串

在JavaScript中，字符串是一个Unicode字符序列（每个字符占16位）。字符串中的字符可以按照索引访问，第一个字符的索引是0。字符串包含在"或'之中，这两种符号都是描述字符串的有效方式。来看看下面的例子：

```
> console.log("Hippopotamus chewing gum");
Hippopotamus chewing gum
> console.log('Single quoted hippopotamus');
Single quoted hippopotamus
> console.log("Broken \n lines");
Broken
 lines
```

最后一行展示了当配合转义字符\使用的时候，有些字符可以成为具有特殊意义的字符。下

面是一些特殊字符的例子。

- `\n`：换行符
- `\t`：制表符
- `\b`：退格
- `\r`：回车
- `\\`：反斜杠
- `\'`：单引号
- `\"`：双引号

JavaScrript字符串默认支持特殊字符以及Unicode字符：

```
> '\xA9'
'©'
> '\u00A9'
'©'
```

就JavaScript中的String、Number和Boolean类型而言，重要的是要知道它们都可以通过包装其原始值（primitive equivalent）而形成对应的包装对象（wrapper object）。下面的示例展示了包装对象的用法：

```
var s = new String("dummy"); //创建一个String对象
console.log(s); //"dummy"
console.log(typeof s); //"object"
var nonObject = "1" + "2"; //创建一个原始的String类型
console.log(typeof nonObject); //"string"
var objString = new String("1" + "2"); ///创建一个String对象
console.log(typeof objString); //"object"
//工具函数
console.log("Hello".length); //5
console.log("Hello".charAt(0)); //"H"
console.log("Hello".charAt(1)); //"e"
console.log("Hello".indexOf("e")); //1
console.log("Hello".lastIndexOf("l")); //3
console.log("Hello".startsWith("H")); //true
console.log("Hello".endsWith("o")); //true
console.log("Hello".includes("X")); //false
var splitStringByWords = "Hello World".split(" ");
console.log(splitStringByWords); //["Hello", "World"]
var splitStringByChars = "Hello World".split("");
console.log(splitStringByChars); //["H", "e", "l", "l", "o", " ",
  "W", "o", "r", "l", "d"]
console.log("lowercasestring".toUpperCase()); //"LOWERCASESTRING"
console.log("UPPPERCASESTRING".toLowerCase()); //"upppercasestring"
console.log("There are no spaces in the end      ".trim());
  //"There are no spaces in the end"
```

JavaScript也允许多行字符串，这种字符串是由` 字符（重音符：https://en.wikipedia.org/wiki/Grave_accent）包围形成的。来看下面的例子：

```
> console.log(`string text on first line
string text on second line `);
"string text on first line
string text on second line "
```

这种字符串也被称为模板字符串（template string），可用于字符串插值（string interpolation）。JavaScript利用这种语法来实现Python风格的字符串插值。

通常，可以像下面这样做：

```
var a=1, b=2;
console.log("Sum of values is :" + (a+b) + " and multiplication is :" + (a*b));
```

如果借助字符串插值的话，代码会变得更清晰：

```
console.log(`Sum of values is :${a+b} and multiplication is : ${a*b}`);
```

6. Undefined

JavaScript使用两个特殊的值来表示不存在有意义的值——null和undefined。前者表明无值（non-value），是有意为之的；而后者表明变量尚未赋值。来看下面的例子：

```
> var xl;
> console.log(typeof xl);
undefined
> console.log(null==undefined);
true
```

7. Boolean

JavaScript中Boolean类型的原始值是由关键字`true`和`false`描述的。下面的规则说明了什么是false，什么是true：

- false、0、空串（""）、NaN、null和undefined都被视为false
- 其他的都被视为true

Boolean类型难以捉摸之处，主要在于其行为与创建之时大相径庭。

在JavaScript中有两种方法可以创建Boolean类型的值。

- 可以将字面量true或false赋给变量。考虑下面的例子：

```
var pBooleanTrue = true;
var pBooleanFalse = false;
```

- 使用`Boolean()`函数。这是个普通的函数，可以返回一个Boolean类型的原始值：

```
var fBooleanTrue = Boolean(true);
var fBooleanFalse = Boolean(false);
```

这两种方法都可以返回所期望的**真值**或**假值**，但如果使用new操作符创建一个Boolean对象，那可就大错特错了。

当使用new操作符和Boolean(value)构造函数时，得到的并不是原始的true或false，而是一个对象，而且不幸的是，JavaScript将对象视为真（truthy）：

```
var oBooleanTrue = new Boolean(true);
var oBooleanFalse = new Boolean(false);
console.log(oBooleanTrue); //true
console.log(typeof oBooleanTrue); //object
if(oBooleanFalse){
  console.log("I am seriously truthy, don't believe me");
}
>"I am seriously truthy, don't believe me"

if(oBooleanTrue){
  console.log("I am also truthy, see ?");
}
>"I am also truthy, see ?"

//使用valueOf()在布尔对象中提取真正的值
if(oBooleanFalse.valueOf()){
  console.log("With valueOf, I am false");
}else{
  console.log("Without valueOf, I am still truthy");
}
>"Without valueOf, I am still truthy"
```

所以，聪明的做法是避免使用Boolean构造函数来创建新的Boolean对象。它破坏了Boolean类型的基本逻辑，应该离这种难以调试的问题代码远远的。

8. instanceof操作符

使用引用类型存储值的一个问题是typeof操作符，不管引用的是什么类型的对象，该操作符返回的都是object。可以使用instanceof操作符来解决该问题。来看下面的例子：

```
var aStringObject = new String("string");
console.log(typeof aStringObject);         //"object"
console.log(aStringObject instanceof String);     //true
var aString = "This is a string";
console.log(aString instanceof String);      //false
```

第3行返回的是false。当讲到原型链的时候，我们会讨论为什么会出现这种情况。

9. Date对象

JavaScript并没有日期数据类型。不过，可以在应用程序中使用Date对象及其方法来处理日期和时间。Date对象中的内容非常详尽，其中包含的一些方法能够处理绝大部分与日期和时间相关的问题。

JavaScript处理日期的方法和Java类似。在JavaScrpt中，日期被存储为自1970年1月1日00:00:00起的毫秒数。

可以使用以下声明创建一个Date对象：

```
var dataObject = new Date([parameters]);
```

Date对象构造函数的参数如下。

- 不提供参数，创建的是当前的日期和时间。例如，`var today = new Date();`。
- 描述日期的字符串，形如`Month day, year hours:minutes:seconds`。例如，`var twoThousandFifteen = new Date("December 31, 2015 23:59:59");`。如果忽略了时、分或秒，这些值会被设置为0。
- 一组表示年、月、日的整数。例如，`var christmas = new Date(2015, 11, 25);`。
- 一组表示年、月、日、时、分和秒的整数。例如，`var christmas = new Date(2015, 11, 25, 21, 00, 00);`。

在JavaScript中创建及处理日期的示例如下：

```
var today = new Date();
console.log(today.getDate()); //27
console.log(today.getMonth()); //4
console.log(today.getFullYear()); //2015
console.log(today.getHours()); //23
console.log(today.getMinutes()); //13
console.log(today.getSeconds()); //10
//自1970年1月1日00:00:00 UTC之后的毫秒数
console.log(today.getTime()); //1432748611392
console.log(today.getTimezoneOffset()); //-330 Minutes

//计算逝去的时间
var start = Date.now();
//循环很长一段时间
for (var i=0;i<100000;i++);
var end = Date.now();
var elapsed = end - start; // 以毫秒为单位的逝去时间
console.log(elapsed); //71
```

对于任何需要对日期和时间进行精细控制的应用程序，推荐使用Moment.js（https://github.com/moment/moment）、Timezone.js（https://github.com/mde/timezone-js）或date.js（https://github.com/MatthewMueller/date）这样的库。这些库为你简化了大量的重复性任务，有助于你将精力集中在其他重要的方面。

10. +操作符

当用作单目操作符的时候，+操作符不会对Number类型产生影响。但如果应用在字符串类型上，会将其转换成数字：

```
var a=25;
a=+a; //对a的值没有影响
console.log(a); //25

var b="70";
console.log(typeof b); //string
b=+b; //将字符串转换成数字
console.log(b); //70
console.log(typeof b); //number
```

程序员通常使用+操作符快速地将字符串转换成数字。但如果字符串字面量无法转换成数字的话，结果会有些出人意料：

```
var c="foo";
c=+c; //将foo转换成数字
console.log(c); //NaN
console.log(typeof c); //number

var zero="";
zero=+zero; //将空串转换成0
console.log(zero);
console.log(typeof zero);
```

随后的章节中会讨论+操作符在其他数据类型上的效果。

11. ++和--操作符

++操作符和--操作符都是一种快捷方式，前者可用于给数值加1，后者可用于从数值中减1。Java和C中都有同样的操作符，大多数人对其并不陌生。考虑下面的代码：

```
var a= 1;
var b= a++;
console.log(a); //2
console.log(b); //1
```

怎么回事？变量b的值难道不应该是2吗？++和--操作符都是单目操作符，既可作为前缀，也可以作为后缀。它们所出现的位置很重要。当++作为前缀的时候，如++a，会先增加变量的值，然后再将该值从表达式返回，而不是像a++那样，先返回a的当前值，然后再增加。来看看下面的代码：

```
var a= 1;
var b= ++a;
console.log(a); //2
console.log(b); //2
```

很多程序员喜欢使用链式赋值的方式将一个值赋给多个变量：

```
var a, b, c;
a = b = c = 0;
```

这样做没有问题，因为赋值操作符（=）的作用就是赋值。在这个例子中，c=0的求值结果为0；这会产生b=0，该表达式的结果也是0，随之就是对a=0求值了。

但如果对上面的例子做一个细小的改动，就会造成截然不同的效果。考虑如下代码：

```
var a = b = 0;
```

在这个例子中，使用var声明的变量只有a，变量b意外地成为了全局变量（如果处于严格模式的话，此处会报错）。想要用好JavaScript，想实现期望的结果时一定要谨慎。

12. 布尔操作符

JavaScript中有三个布尔操作符：AND（&）、OR（|）和NOT（!）。

在讨论逻辑AND和OR操作符之前，需要搞明白它们是如何产生布尔结果的。逻辑操作符的求值方向是从左到右，使用下面的短路规则进行测试。

- **逻辑AND**：如果第一个操作数能够决定最终结果，那么第二个操作数就无需再求值了。

  在下面的例子中，凡是表达式的右侧按短路求值规则得以执行的，我都做了突出标记：

```
console.log(true && true); // true AND true返回true
console.log(true && false);// true AND false返回false
console.log(false && true);// false AND true返回false
console.log("Foo" && "Bar");// Foo(true) AND Bar(true)返回Bar
console.log(false && "Foo");// false && Foo(true)返回false
console.log("Foo" && false);// Foo(true) && false返回false
console.log(false && (1 == 2));// false && false(1==2)返回false
```

- **逻辑OR**：如果第一个操作数为真，第二个操作数就无需再求值了。

```
console.log(true || true); // true AND true返回true
console.log(true || false);// true AND false返回true
console.log(false || true);// false AND true返回true
console.log("Foo" || "Bar");// Foo(true) AND Bar(true)返回Foo
console.log(false || "Foo");// false && Foo(true)返回Foo
console.log("Foo" || false);// Foo(true) && false返回Foo
console.log(false || (1 == 2));// false && false(1==2)返回false
```

  不过，逻辑AND和逻辑OR也可以用于非Boolean类型的操作数。如果左、右操作数有任意一个不是原始的Boolean类型值，AND和OR也不会返回Boolean类型值。

现在我们来解释这三个逻辑布尔运算符。

- 逻辑AND（&&）：如果第一个操作数为假，则它返回该操作数；如果为真，则返回第二个操作数。

```
console.log (0 && "Foo"); //第一个操作数为假，则返回该操作数
console.log ("Foo" && "Bar"); //第一个操作数为真，则返回第二个操作数
```

- 逻辑OR（||）：如果第一个操作数为真，它返回该操作数；否则，返回第二个操作数。

  ```
  console.log (0 || "Foo"); //第一个操作数为假，则返回第二个操作数
  console.log ("Foo" || "Bar"); //第一个操作数为真，则返回该操作数
  console.log (0 || false); //第一个操作数为假，则返回第二个操作数
  ```

  逻辑OR的典型用法是为变量分配默认值：

  ```
  function greeting(name){
    name = name || "John";
    console.log("Hello " + name);
  }

  greeting("Johnson"); // 输出"Hello Johnson";
  greeting(); //输出"Hello John"
  ```

  你会在多数具有专业水准的JavaScript库中频繁地看到这种用法。你应该理解如何通过使用逻辑OR运算符进行默认赋值。

- 逻辑NOT：总是返回一个Boolean类型的值。所返回的具体值如下：

  ```
  //如果操作数是对象，返回false
  var s = new String("string");
  console.log(!s); //false

  //如果操作数是数字0，返回true
  var t = 0;
  console.log(!t); //true

  //如果操作数是除0之外的任意数字，返回false
  var x = 11;
  console.log(!x); //false

  //如果操作数是null或NaN，返回true
  var y =null;
  var z = NaN;
  console.log(!y); //true
  console.log(!z); //true
  //如果操作数未定义（undefined），返回true
  var foo;
  console.log(!foo); //true
  ```

除此之外，JavaScript也支持类似于C中的三元操作符：

```
var allowedToDrive = (age > 21) ? "yes" : "no";
```

如果`(age > 21)`为真，`?`之后的值会被赋给变量`allowedToDrive`；否则，将`:`之后的值赋给该变量。其效果等同于if-else条件语句。来看另外一个例子：

```
function isAllowedToDrive(age){
  if(age>21){
    return true;
  }else{
```

```
    return false;
  }
}
console.log(isAllowedToDrive(22));
```

在这个例子中，函数`isAllowedToDrive()`接受一个整型参数`age`。根据该参数的值，向调用函数返回true或false。这是一种众所周知，也是最为熟悉的if-else条件逻辑。大多数时候，if-else能够使得代码更易读。对于单一条件而言，也可以使用三元操作符，但如果发现自己在比较复杂的表达式中用到了该操作符，最好还是换用if-else，因为相较于难懂的三元操作符表达式，if-else的易读性更好。

if-else条件语句可以嵌套如下：

```
if (condition1) {
  statement1
} else if (condition2) {
  statement2
} else if (condition3) {
  statement3
}
..
} else {
  statementN
}
```

也可以纯粹依据个人喜好，像下面这样缩进嵌套的`else if`：

```
if (condition1) {
  statement1
} else
    if (condition2) {
```

不要在条件语句中进行赋值。大多数时候，这都是错误的用法：

```
if(a=b) {
  //执行相关操作
}
```

这多半是个错误，代码本应该是`if(a==b)`，或者写得更好些，`if(a===b)`。如果把条件语句误写成赋值语句，算是给自己留下了一个极难查找的错误。如果真的想在if语句中使用赋值语句，务必理清思路。

一种方法是在赋值语句外多加上一对括号：

```
if((a=b)){
  //确定执行相关操作
}
```

还可以使用switch-case语句来处理条件执行。JavaScript中的switch-case与C或Java中的类似。来看下面的例子：

```
function sayDay(day){
  switch(day){
    case 1: console.log("Sunday");
      break;
    case 2: console.log("Monday");
      break;
    default:
      console.log("We live in a binary world. Go to Pluto");
  }
}

sayDay(1); //Sunday
sayDay(3); //We live in a binary world. Go to Pluto
```

这种结构存在的问题是必须在每一个case分支中加入`break`，否则语句会持续执行到下一级。如果把第一个case语句中的`break`语句移去，则会输出以下结果：

```
>sayDay(1);
Sunday
Monday
```

如你所见，如果不使用`break`语句立刻跳出所满足的条件，就会接着执行下一级case分支中的语句。这种错误很难检测。但如果打算直落到下一级分支的话，有一种常见的条件逻辑写法：

```
function debug(level,msg){
  switch(level){
    case "INFO": //此处有意忽略break语句
    case "WARN" :
    case "DEBUG": console.log(level+ ": " + msg);
      break;
    case "ERROR": console.error(msg);
  }
}
debug("INFO","Info Message");
debug("DEBUG","Debug Message");
debug("ERROR","Fatal Exception");
```

在这个例子中，出于精简switch-case的目的，我们有意让执行流程直落下来（fall-through）。如果参数level的值是INFO、WARN或DEBUG，利用switch-case使得流程落在单一的执行位置上。为了实现这个目的，我们省去了`break`语句。如果想模仿这种switch语句的写法，一定要记得将用法通过文档记录下来，以确保更好的可读性。

switch语句可以有一个`default`分支，用于处理其他分支所不能覆盖的情况。

JavaScript还拥有while和do-while循环。while循环可以在满足条件之前迭代一组表达式。下面第一个例子迭代了`{}`中的语句，直到满足表达式`i<10`为止。记住，如果变量`i`的值已经大于`10`，则循环一次都不会执行。

```
var i=0;
while(i<10){
```

```
  i=i+1;
  console.log(i);
}
```

下面的循环会永无休止地执行下去，因为条件总是成立，这会导致灾难性的后果。这样的程序会耗尽所有的内存或其他资源：

```
//死循环
while(true){
  //不停地执行此处的操作
}
```

如果想确保循环至少执行一次，可以使用do-while循环（有时也称为后置条件循环）：

```
var choice;
do {
  choice=getChoiceFromUserInput();
} while(!isInputValid(input));
```

在这个例子中，从用户处获得输入，直到输入内容有效为止。如果用户输入的内容不符合要求，会要求用户继续输入。通常认为，从逻辑上来说，所有的do-while循环都可以改写成while循环。但是在刚刚看到的那个例子中，就非常适合使用do-while循环，因为我们希望在执行过一次循环之后再检查条件。

和C或Java类似，JavaScript也有一个功能强大的循环——for循环。for循环流行的原因在于只用一行就可以定义循环控制条件。

下面的例子可以打印出5次`Hello`：

```
for (var i=0;i<5;i++){
  console.log("Hello");
}
```

在循环定义中，循环计数器`i`的初始值被设为`0`，循环退出条件设为`i<5`，最后定义的是计数器增量。

上例中的三个表达式都是可选的。如果需要的话，是可以忽略不写的。例如，下面的for循环尽管形式不同，但生成的结果是相同的：

```
var x=0;
//忽略初始化
for (;x<5;x++){
  console.log("Hello");
}

//忽略退出条件
for (var j=0;;j++){
  //退出条件
  if(j>=5){
```

```
    break;
  }else{
    console.log("Hello");
  }
}
//忽略修改循环变量
for (var k=0; k<5;){
  console.log("Hello");
  k++;
}
```

也可以将这三个表达式全部忽略。一种常用的惯用写法是使用空循环体。下面的循环用于将数组中所有的元素值设为100。注意，这个for循环并没有循环体：

```
var arr = [10, 20, 30];
// 给所有的数组元素赋值100
for (i = 0; i < arr.length; arr[i++] = 100);
console.log(arr);
```

空循环体的位置就在for循环语句之后。增量表达式同样修改了数组内容。本书随后会讨论数组，在这里只需要知道循环定义本身将所有的数组元素都设置成100就够了。

13. 相等

JavaScript提供两种相等关系：严格（strict）相等和非严格（loose）相等。从根本上而言，在比较两个值的时候，非严格相等会进行类型转换，而严格相等不进行类型转换。可以通过`===`执行严格相等比较，通过`==`执行非严格相等比较。

ECMAScript 6还提供了`Object.is`方法来进行类似于`===`的严格相等比较，但是`Object.is`会针对NaN做特别处理：–0和+0。当NaN===NaN和NaN==NaN的结果都为false时，`Object.is(NaN, NaN)`将返回true。

(1) 使用`===`的严格相等

严格相等在比较两个值的时候，不会进行任何隐式类型转换。比较过程中会应用以下规则。

- 如果值的类型不同，则不相等。
- 对于相同类型的非数值，如果值相同的话，则相等。
- 对于Number类型，用严格相等来比较值。如果值相等，则结果为true。但是NaN不等于任何数字，`NaN===<a number>`的结果为`false`。

严格相等是检查相等关系时应该使用的正确方法。应该设立这么一条规则：坚持使用`===`，避免使用`==`。

条　　件	输　　出
`"" === "0"`	false
`0 === ""`	false
`0 === "0"`	false
`false === "false"`	false
`false === "0"`	false
`false === undefined`	false
`false === null`	false
`null === undefined`	false

如果比较对象的话，可以得到如下结果：

条　　件	输　　出
`{} === {}`	false
`new String('bah') === 'bah';`	false
`new Number(1) === 1;`	false
`var bar = {}; bar === bar;`	true

下面这些例子，你应该在JS Bin或Node REPL中试验一下：

```
var n = 0;
var o = new String("0");
var s = "0";
var b = false;

console.log(n === n); // true - 数值相同
console.log(o === o); // true - 非数字类型比较
console.log(s === s); // true - 同上

console.log(n === o); // false - 不进行隐式类型转换，因此类型不相同
console.log(n === s); // false - 类型不同
console.log(o === s); // false - 类型不同
console.log(null === undefined); // false
console.log(o === null); // false
console.log(o === undefined); // false
```

在进行严格相等比较的时候，可以使用`!==`来处理**不相等**的情况。

(2) 使用`==`的非严格相等

不要使用这种形式的相等比较。说真的，离它越远越好。非严格相等有很多缺点，其主要原因在于JavaScript的弱类型化。相等操作符`==`在比较之前会先尝试转换类型。下面的例子展示了这个过程：

条　　件	输　　出
`"" == "0"`	false
`0 == ""`	true

（续）

条　件	输　出
`0 == "0"`	true
`false == "false"`	false
`false == "0"`	true
`false == undefined`	false
`false == null`	false
`null == undefined`	true

从这些例子中显然可以看出非严格相等会导致出人意料的结果。另外，隐式类型转换也会对性能产生影响。因此在JavaScript中，一般不要使用非严格相等。

1.3.2 JavaScript 类型

我们简要地提过，JavaScript是一种动态语言。如果你之前有强类型语言（如Java）的使用经验，可能会有点不适应完全没有类型检查的情况。JavaScript的忠实粉丝认为，这门语言是拥有**标签**（tag）或者说是**子类型**（subtype）的，但没有类型。尽管JavaScript并没有传统定义上的类型，但是绝对有必要搞明白，JavaScript在内部是如何处理数据类型并进行强制转换（coercion）的。所有正式的JavaScript程序都少不了要以某种形式来处理值的强制转换，因此理解相关的概念很重要。

当自己动手修改类型的时候会发生显式强制转换（explicit coercion）。在下面的例子中，使用`toString()`将一个数字转换成了String类型，并从中提取出第二个字符：

```
var fortyTwo = 42;
console.log(fortyTwo.toString()[1]); //打印出"2"
```

这是一个显式强制转换的例子。我们谈及的**类型**一词并不是严格意义上的，因为在声明变量`fortyTwo`的时候，不需要强制指定类型。

不过，这种强制转换还可以通过很多其他方式出现。显式强制转换易于理解，可靠性也好；但如果不注意的话，强制类型转换会产生非常出人意料的结果。

强制转换方面的困惑可能是最让JavaSript开发人员沮丧的话题之一。为了确保你不会产生这样的疑惑，来回顾一下JavaScript中的类型。之前讨论过这方面的一些概念：

```
typeof 1              === "number";    // true
typeof "1"            === "string";    // true
typeof { age: 39 }    === "object";    // true
typeof Symbol()       === "symbol";    // true
typeof undefined      === "undefined"; // true
typeof true           === "boolean";   // true
```

到目前为止，一切都还不错。我们对此已经有过了解，刚刚看到的例子再次巩固了关于类型的概念。

将值从一种类型转换成另一种类型叫作**类型转换**（casting）或显式强制转换。JavaScript也会基于某些猜测来更改值的类型，这叫作隐式强制转换（implict coercion）。这种猜测使得JavaScript解决了一些问题，但同时也遗憾地引发了一些悄无声息且出乎意料的错误。下面的代码片段展示了一些显式和隐式的强制转换：

```
var t=1;
var u=""+t; //隐式强制转换
console.log(typeof t); //"number"
console.log(typeof u); //"string"
var v=String(t); //显式强制转换
console.log(typeof v); //"string"
var x=null
console.log(""+x); //"null"
```

很容易就可以看出结果。当`t`的值是数字（这里是`1`）的时候，`""+t`会使得JavaScript认为你要将某些内容和字符串`""`拼接在一起。因为只有字符串才能彼此拼接，所以JavaScript就先将数字`1`转换成字符串`"1"`，然后再将两者拼接起来，形成最终的结果（字符串值）。这就是JavaScript实施隐式转换的过程。而`String(t)`则是一个意图非常明确的调用，用于将一个数字转换成String类型。这是一种显式类型转换。最后一行代码的结果有些让人意外。我们将`null`和`""`拼接在了一起，居然没有出错？

那么，JavaScript是怎么做类型转换的？一个抽象值是怎么变成字符串、数字或者布尔值的？其实，JavaScript在内部是依靠`toSting()`、`toNumber()`和`toBoolean()`方法来实现这一切的。

当一个非String类型的值被强制转换成String类型时，JavaScript在内部使用的是`toString()`方法。所有的原始值都有对应的字符串形式——null的字符串形式是`"null"`，undefined的字符串形式是`"undefined"`，等等。对于Java开发人员而言，这就好比是一个拥有`toString()`方法的类，该方法能够返回类的字符串描述。在讨论对象的时候，我们将会看到这究竟是如何实现的。

因此，可以实现类似于下面的操作：

```
var a="abc";
console.log(a.length);
console.log(a.toUpperCase());
```

如果在敲入这些代码片段时稍加留心，你会意识到一些奇怪的地方。我们怎么能在原始值上调用属性和方法呢？原始值怎么会有对象才有的属性和方法呢？它们其实并没有。

之前讨论过，JavaScript默认会很贴心地将原始值包裹起来形成包装对象，这样我们就可以直接访问包装对象的方法和属性，就好像这些方法和属性是原始值自带的一样。

当非数字值需要被强制转换成数字时，JavaScript会在内部使用`toNumber()`方法：`true`变成`1`，`undefined`变成NaN，`false`变成`0`，`null`变成`0`。在字符串上使用`toNumber()`时会进行字面上的转换，如果转换失败，则返回`NaN`。

那么，下面这个例子呢？

```
typeof null ==="object" //true
```

null是对象？是的，这是由于一个历史悠久的bug导致的。因为这个bug，在测试某个值是否为null的时候一定要小心：

```
var x = null;
if (!x && typeof x === "object"){
  console.log("100% null");
}
```

对于其他有类型的东西，如函数呢？

```
f = function test() {
  return 12;
}
console.log(typeof f === "function"); //打印 "true"
```

数组呢？

```
console.log(typeof [1.2.3.4]); // "object"
```

它们铁定也是对象，本书随后会详细地讨论函数和数组。

在JavaScript中，值有类型，变量没有。由于JavaScript的动态性质，变量可以随时保存任何类型的值。

JavaScript并不强制类型，也就是说语言本身并不要求变量中值的类型必须一直和初始类型相同。变量中可以保存字符串，下一次赋值时可以保存数字，等等。

```
var a = 1;
typeof a; // "number"
a = false;
typeof a; // "boolean"
```

`typeof`操作符总是返回String类型：

```
typeof typeof 1; // "string"
```

1.3.3 自动插入分号

尽管JavaScript的语法风格与C类似，但它并不强制要求在源代码中使用分号。

但这并不是说JavaScript就不需要分号，JavaScript语言解析器需要分号来理解源代码。因此，解析器在遇到由于缺失分号而导致的解析错误时会自动插入分号。重要的是要注意，**自动插入分**

号（automatic semicolon insertion，ASI）只在有换行符的时候才会发生。分号并不会插入到一行代码的中间。

基本上来说，如果JavaScript解析器在解析某一行的时候出现了解析错误（丢失了需要的分号），而且可以插入分号，那它就会这么做。插入分号的标准是什么呢？答案是：仅当某些语句的结尾和换行符之间只有空白字符或注释的时候，才会插入分号。

关于ASI有过激烈的争论。无可非议，这个特性是一个非常糟糕的设计选择。互联网上曾进行过激烈的讨论，比如https://github.com/twbs/bootstrap/issues/3057以及https://brendaneich.com/2012/04/theinfernal-semicolon/。

在辨别这些争论正确与否之前，得先明白ASI产生了哪些影响。下面的这些语句会受到ASI的影响：

- 空语句
- var语句
- 表达式语句
- do-while语句
- continue语句
- break语句
- return语句
- throw语句

ASI的理念是使得行尾的分号成为可选的，这样，ASI就能够帮助解析器判断一个语句什么时候结束。通常，语句以分号作为结尾。ASI规定一个语句在下列情况下也可以结束：

- 行终结符（例如换行符）之后是一个非法的词法单元（token）
- 遇到右大括号
- 抵达文件尾部

来看下面的例子：

```
if (a < 1) a = 1 console.log(a)
```

出现在`1`后面的词法单元`console`是非法的，因此触发了ASI：

```
if (a < 1) a = 1; console.log(a);
```

在下面的代码中，大括号中语句的终结符不是分号：

```
function add(a,b) { return a+b }
```

ASI会对上面的代码生成一个语法正确的版本：

```
function add(a,b) { return a+b; }
```

1.3.4 JavaScript 代码风格指南

每一门编程语言都会形成自己的一套代码风格和结构。遗憾的是，开发新手并没有花费很多精力学习语言风格方面的细微差异。一旦养成坏毛病，就很难习得这项技能了。要想写就优美、易读以及可维护性高的代码，重要的是要学会正确的代码风格。风格方面的建议可谓层出不穷，我们只挑选那些实用性最强的。我们会在合适的时候讨论适合的风格。让我们先来确定风格方面的一些基本规则。

1. 空白字符

尽管空白字符在JavaScript中并不重要，但是正确使用这些字符能够提高代码的可读性。下面的操作指南有助于管理代码中的空白字符。

- 绝对不要混用空格和制表符。
- 在编码之前，选择到底是使用软缩进（空格）还是真实的制表符。考虑到可读性，推荐将编辑器的缩进距离设置成两个字符，这意味着两个空格或是代表一个真实的制表符的两个空格。
- 确保"显示隐藏字符"（show invisibles）设置选项一直是启用的。这样做的好处如下：
  - 强制一致性
  - 消除行尾的空白字符
  - 消除空行和空白字符
  - 易于识别差异
  - 尽可能使用EditorConfig（http://editorconfig.org/）

2. 括号、换行符和大括号

在if、else、while和try中坚持使用空格和大括号，并将这些语句分布在多行中。这种风格提倡的是可读性。来看看下面的代码：

```
// 拥挤的书写风格（不良）
if(condition) doSomeTask();

while(condition) i++;

for(var i=0;i<10;i++) iterate();

// 利用空白字符提高可读性（优良）
// 在领头的括号前加上一个空格
if (condition) {
  // 语句
}

while ( condition ) {
```

```
  // 语句
}

for ( var i = 0; i < 100; i++ ) {
  // 语句
}

// 更好的写法:

var i,
    length = 100;

for ( i = 0; i < length; i++ ) {
  // 语句
}

// 或者……

var i = 0,
    length = 100;

for ( ; i < length; i++ ) {
  // 语句
}

var value;

for ( value in object ) {
  // 语句
}

if ( true ) {
  // 语句
} else {
  // 语句
}

// 没有在操作符两边加上空格
// 不良的写法
var x=y+5;

// 优良的写法
var x = y + 5;

// 使用单个换行符结束文件
// 不良的写法
(function(global) {
  // ...stuff...
})(this);

// 不良的写法
(function(global) {
  // ...stuff...
})(this);↵
↵
```

```
// 优良的写法
(function(global) {
  // ...stuff...
})(this);↲
```

3. 引号

不管你喜欢单引号还是双引号都没关系，JavaScript解析器并不会对这两者区别对待。但出于一致性的考虑，切记不要在同一个项目中混用这两种引号，请选择并坚持一种风格。

4. 行尾和空行

空白字符会使得代码的差异与变更无法辨识。很多编辑器允许自动移除多余的空行及行尾字符，你应该使用这项功能。

5. 类型检查

可以像下面这样检查变量的类型：

```
//String:
typeof variable === "string"
//Number:
typeof variable === "number"
//Boolean:
typeof variable === "boolean"
//Object:
typeof variable === "object"
//null:
variable === null
//null或undefined:
variable == null
```

6. 类型转换

在语句的一开始就执行强制类型转换：

```
// 不良
const totalScore = this.reviewScore + '';
// 优良
const totalScore = String(this.reviewScore);
```

对Number类型使用`parseInt()`的时候，总是加入基数用于类型转换：

```
const inputValue = '4';
// 不良
const val = new Number(inputValue);
// 不良
const val = +inputValue;
// 不良
```

```
const val = inputValue >> 0;
// 不良
const val = parseInt(inputValue);
// 优良
const val = Number(inputValue);
// 优良
const val = parseInt(inputValue, 10);
```

下面的例子展示了如何使用Boolean()进行类型转换：

```
const age = 0; // 不良
const hasAge = new Boolean(age); // 优良
const hasAge = Boolean(age); // 优良
const hasAge = !!age;
```

7. 条件求值

JavaScript中有各种各样关于条件语句的风格指南。来研究一下下面的代码：

```
// 当数组长度不为空时，
// 不良写法：
if ( array.length > 0 ) ...

// 测试逻辑真（优良的写法）：
if ( array.length ) ...

// 当数组长度为空时，
// 不良写法：
if ( array.length === 0 ) ...

// 测试逻辑真（优良的写法）：
if ( !array.length ) ...

// 检查字符串是否为空时，
// 不良写法：
if ( string !== "" ) ...

// 测试逻辑真（优良的写法）：
if ( string ) ...

// 检查字符串是否为空时，
// 不良写法：
if ( string === "" ) ...

// 测试逻辑假（优良的写法）：
if ( !string ) ...

// 检查引用是否有效时，
// 不良写法：
if ( foo === true ) ...

// 优良的写法：
if ( foo ) ...
```

```
// 检查引用是否无效时，
// 不良写法：
if ( foo === false ) ...

// 优良的写法：
if ( !foo ) ...

// 这样写的话，0、""、null、undefined、NaN也能够满足条件
// 如果你必须针对false测试，可以使用：
if ( foo === false ) ...

// 引用可能会是null或undefined，但绝不会是false、""或0，
// 不良写法：
if ( foo === null || foo === undefined ) ...

// 优良的写法：
if ( foo == null ) ...

// 别把事情复杂化
return x === 0 ? 'sunday' : x === 1 ? 'Monday' : 'Tuesday';

// 这样写更好：
if (x === 0) {
    return 'Sunday';
} else if (x === 1) {
    return 'Monday';
} else {
    return 'Tuesday';
}

// 锦上添花的写法：
switch (x) {
    case 0:
        return 'Sunday';
    case 1:
        return 'Monday';
    default:
        return 'Tuesday';
}
```

8. 命名

命名极为重要。我打包票你肯定碰到过一些在命名方面惜字如金且令人难以琢磨的代码。来看下面这些代码：

```
// 别用单个字母命名。使用描述性的名字
// 差劲的命名
function q() {
}

// 不错的命名
function query() {
```

```
}

// 对于对象、函数和实例，使用驼峰命名法
// 差劲的命名
const OBJEcT = {};
const this_is_object = {};
function c() {}

// 不错的命名
const thisIsObject = {};
function thisIsFunction() {}

// 对于构造函数和类，使用Pascal命名法
// 差劲的命名
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// 不错的命名
class User {
  constructor(options) {
    this.name = options.name;
  }
}
const good = new User({
  name: 'yup',
});

// 命名私有属性的时候，在名字前面加上下划线_
// 差劲的命名
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// 不错的命名
this._firstName = 'Panda';
```

9. 邪恶的`eval()`

作为JavaScript中被误用的最为严重的方法之一，`eval()`方法可以接受一个包含JavaScript代码的字符串，然后编译并执行。`eval()`仅适合在少数情况下使用，比如基于用户输入来构建表达式。

大多数情况下，使用`eval()`的原因仅仅是因为它能把活儿给干了。`eval()`方法的技巧性太强，会使得代码不可预测。它运行速度缓慢、难以处理，如果你不小心犯错，往往还会加重损害。如果考虑使用`eval()`的话，那么也许会有其他更好的解决方法。

下面的代码片段展示了`eval()`的用法：

```
console.log(typeof eval(new String("1+1"))); // "object"
console.log(eval(new String("1+1")));        //1+1
console.log(eval("1+1"));                    // 2
console.log(typeof eval("1+1"));             // 返回"number"
var expression = new String("1+1");
console.log(eval(expression.toString()));    //2
```

我不打算再展示`eval()`的更多用法，这个方法不鼓励使用，离它远点。

10. 严格模式

ECMAScript 5中的严格模式（strict mode）能够产生更清晰的代码，该模式减少了一些不安全的特性，增加了更多的警告信息，代码行为也更符合逻辑。普通（非严格）模式也叫作**懒散模式**（sloppy mode）。严格模式能够帮助你避免一些不够细致的编程实践。如果刚入手一个新的JavaScript项目，强烈建议一开始就使用严格模式。

可以先在JavaScript文件或`<script>`元素中输入下面的一行代码来进入严格模式：

```
'use strict';
```

注意，不支持ECMAScript 5的JavaScript引擎会简单地忽略上面的语句，继续在非严格模式下运行。

如果想以函数为单位启用严格模式，可以像下面这样做：

```
function foo() {
    'use strict';
}
```

当处理那些可能无法在严格模式下运行的遗留代码的时候，这么做非常方便。

如果使用了已有的遗留代码，一定要小心，因为严格模式在这种情况下会造成麻烦。对此有一些注意事项。

(1) 对已有的代码启用严格模式会出问题

代码有可能会依赖于某个已经不再可用的特性，或是某种在懒散模式和严格模式中存在差异的行为。别忘了可以在处于懒散模式的JavaScript文件中加入一个处于严格模式的函数。

(2) 谨慎打包

当合并或压缩JavaScript文件时，要注意严格模式并不会在预期的位置上打开或关闭，这会导致代码出问题。

接下来将更为详细地解释严格模式的一些特性。这些特性通常并不需要了解，因为在你进行不当操作的时候，基本上都会收到警告信息。

(3) 严格模式中的变量必须声明

在严格模式中，所有的变量必须明确声明，这有助于避免输入错误。在懒散模式中，对未声明的变量赋值会生成一个全局变量：

```
function sloppyFunc() {
  sloppyVar = 123;
}
sloppyFunc(); // 创建全局变量sloppyVar
console.log(sloppyVar); // 123
```

在严格模式中，对未声明的变量赋值会抛出异常：

```
function strictFunc() {
  'use strict';
  strictVar = 123;
}
strictFunc(); //ReferenceError: strictVar is not defined
```

(4) 严格模式中的`eval()`函数行为更清晰

在严格模式中，`eval()`函数的行为变得不再那么古怪：在被求值的字符串（evaluated string）中声明的变量，不会被添加到`eval()`的外围作用域中。

(5) 严格模式中被禁止的特性

严格模式中不允许使用with语句（随后将会讨论该语句）。你会在编译时（载入代码的时候）得到一个语法错误。

在懒散模式中，前面带有0的数字被认为是八进制（基数为8）：

```
> 010 === 8 true
```

在严格模式中，如果使用这种形式的字面量写法，会导致语法错误：

```
function f() {
'use strict';
return 010
}
//SyntaxError: Octal literals are not allowed in
```

11. 运行JSHint

JSHint是一个能够标记出JavaScript程序中可疑用法的工具，其核心是由一个库和一个作为Node模块分发的**命令行界面**（command line interface，CLI）程序组成的。

如果安装过Node.js，就可以像下面这样使用`npm`来安装JSHint：

```
npm install jshint -g
```

安装好JSHint之后，就可以用它来检查一个或多个JavaScript文件了。将下面的代码片段保存为test.js文件：

```
function f(condition) {
  switch (condition) {
  case 1:
    console.log(1);
  case 2:
    console.log(1);
  }
}
```

当使用JSHint运行该文件时，它会发出警告，告诉我们switch的case分支中丢失了break语句：

```
>jshint test.js
test.js: line 4, col 19, Expected a 'break' statement before 'case'.
1 error
```

可以对JSHint进行配置以符合个人需要。请参阅相应的文档（http://jshint.com/docs/）来了解如何根据项目实际需求定制JSHint。我个人广泛地使用了JSHint，建议你也开始使用它。你会惊讶于这么一个简单的工具竟然能修复如此多的隐藏bug以及代码风格问题。

你可以在项目的根目录中运行JSHint来检查整个项目。可以将JSHint指令放入.jshinrc文件中。该文件的格式类似于下面这样：

```
{
    "asi": false,
    "expr": true,
    "loopfunc": true,
    "curly": false,
    "evil": true,
    "white": true,
    "undef": true,
    "indent": 4
}
```

1.4 小结

本章，我们学习了JavaScript语法、类型以及代码风格方面的基础知识。本章特意没有讨论函数、变量作用域以及闭包等重要话题，因为后面会用专门的章节进行介绍。相信本章帮你理解了JavaScript中的一些主要概念。掌握了这些基础知识，接下来就要学习如何编写具有专业水准的JavaScript代码了。

第 2 章

函数、闭包与模块

上一章特意没有对JavaScript的某些方面展开讨论。JavaScript的强大与优雅源于这门语言的一些特性。如果你是一名中高级JavaScript开发人员，会积极地使用对象和函数。但在很多时候，那些尚在入门级别蹒跚的开发人员会对JavaScript的核心构件形成一种似是而非甚至错误的理解。很多程序员由于不能很好地掌握JavaScript中函数方面的知识,因此对于闭包的概念理解得很不到位。在JavaScript中，对象、函数和闭包之间存在着紧密的联系，而理解这种联系能够极大地增强你的JavaScript编程技艺，为开发各种应用程序打下坚实的基础。

JavaScript离不开函数。理解了函数，就意味着你的兵器库中多了一件最有力的武器。有关函数最重要的一点就是：在JavaScript中，函数是头等对象（first-class object），其处理方式与其他JavaScript对象无异。和其他JavaScript数据类型一样，函数可以被变量引用，可以使用字面形式声明，甚至可以作为函数参数传递。

如同JavaScript中的其他对象，函数可以：

- 通过字面形式创建
- 可以赋值给变量、数组元素以及其他对象的属性
- 可以作为函数参数传递
- 可以作为函数值返回
- 可以拥有能动态创建和赋值的属性

在本章以及其他章节中，将对JavaScript函数的这些独特功能进行阐述。

2.1 函数的字面形式

作为主要执行单元的函数，是JavaScript中最重要的概念之一。你可以把代码放在一起形成函数，进而为程序添加上结构。

JavaScript函数可以使用函数的字面形式来声明。

函数的字面形式由四部分组成：

- function关键字
- 可选的函数名称，如果指定的话，这个名称必须是一个有效的JavaScript标识符
- 放在括号中的参数名列表。就算没有参数，也得给出一个空括号
- 函数体是包含在大括号中的一系列JavaScript语句

函数声明

下面这个非常简单的例子演示了一个函数声明的所有组成部分：

```
function add(a,b){
  return a+b;
}
c = add(1,2);
console.log(c); // 打印出3
```

函数声明以关键字function开头，后面是函数名称。函数名称是可选的。如果省略了函数名，该函数就称为匿名函数。随后会介绍匿名函数的用法。第三部分是包含在括号中的函数参数，参数名之间用逗号分隔。这些参数被作为函数内部定义的变量，其初始值并非undefined，而是在调用函数时所提供的参数值。第四部分是大括号中的一组语句，这些语句就是函数体，在函数被调用时执行。

这种声明函数的方法也叫作**函数语句**（function statement）。使用这种方法声明函数时，函数的内容会被编译并创建出一个与函数同名的对象。

也可以利用**函数表达式**（function expression）来声明函数：

```
var add = function(a,b){
   return a+b;
}
c = add(1,2);
console.log(c); //打印出3
```

在这里，我们创建了一个匿名函数并将其赋给变量add，该变量随后可用于调用函数。使用函数表达式声明的问题在于无法递归调用这种函数。递归是一种优雅的编码方式，可以使函数调用自身。解决这种限制的方法是采用具名函数表达式（named function expression）。举例来说，下面的函数可用于计算给定数值n的阶乘：

```
var facto = function factorial(n) {
  if (n <= 1)
    return 1;
  return n * factorial(n - 1);
};
console.log(facto(3)); //打印出6
```

在本例中，我们并没有创建匿名函数，而是创建了一个具名函数。因为函数有了名称，所以

自然就可以递归地调用自身了。

最后，还可以创建调用自身的函数表达式（self-invoking function expression，稍后将会讨论）：

```
(function sayHello() {
  console.log("hello!");
})();
```

定义完成，就可以在其他的JavaScript函数中调用该函数了。函数执行完毕后，调用方（执行被调用函数）继续往下执行。也可以将函数作为参数传递给另一个函数：

```
function changeCase(val) {
  return val.toUpperCase();
}
function demofunc(a, passfunction) {
  console.log(passfunction(a));
}
demofunc("smallcase", changeCase);
```

在上面的例子中，调用函数`demoFunc()`时使用了两个参数。第一个参数是要转换成大写的字符串，第二个参数是对函数`changeCase()`的引用。在`demoFunc()`中，是通过传递给`passfunction`参数的引用来调用`changeCase()`函数的。这里，将一个函数引用作为参数传递给了另一个函数。后面学习回调函数的时候，将详细地讲述这种功能强大的概念。

函数可以返回值，也可以不返回值。在之前的例子中，函数`add`向调用者返回了一个值。除了在函数结尾处返回值之外，显式地调用`return`还可以有条件地从函数中返回：

```
var looper = function(x){
  if (x%5===0) {
    return;
  }
  console.log(x)
}
for(var i=1;i<10;i++){
  looper(i);
}
```

这段代码会打印出1、2、3、4、6、7、8、9，但不包括5。当`if (x%5===0)`为真时，代码会从函数返回，不再执行余下的语句。

2.2　函数作为数据

在JavaScript中，函数可以赋给变量，变量被视为数据。你很快就会看到这种概念的强大之处。来看下面的例子：

```
var say = console.log;
say("I can also say things");
```

在上面的例子中，将熟悉的console.log()函数赋给了变量say。任何函数都可以像这样赋给变量。在变量后加上括号就可以调用对应的函数。此外，还可以将函数作为参数传入别的函数。仔细研究下面的例子，并在JS Bin中输入代码：

```
var validateDataForAge = function(data) {
  person = data();
  console.log(person);
  if (person.age <1 || person.age > 99){
    return true;
  }else{
    return false;
  }
};

var errorHandlerForAge = function(error) {
  console.log("Error while processing age");
};

function parseRequest(data,validateData,errorHandler) {
  var error = validateData(data);
  if (!error) {
    console.log("no errors");
  } else {
    errorHandler();
  }
}

var generateDataForScientist = function() {
  return {
    name: "Albert Einstein",
    age : Math.floor(Math.random() * (100 - 1)) + 1,
  };
};

var generateDataForComposer = function() {
  return {
    name: "JS Bach",
    age : Math.floor(Math.random() * (100 - 1)) + 1,
  };
};

//解析请求
parseRequest(generateDataForScientist, validateDataForAge,
errorHandlerForAge);
parseRequest(generateDataForComposer, validateDataForAge,
errorHandlerForAge);
```

在这个例子中，函数作为参数传给了parseRequest()函数。在两次不同的调用中，generateDataForScientist和generateDataForComposers先后作为不同的参数被传入，另外两个函数参数则保持不变。我们定义了一个通用的parseRequest()，它接受三个函数作为参数，负责将特定的部分组合在一起：数据、验证器（validator）和错误处理程序。可以对

parseRequest()函数进行扩展和定制，同时，因为每次发起请求的时候都要调用该函数，所以可以作为一个清晰的调试点（debugging point）。我确信你已经开始认识到JavaScript函数所提供的强大功能了。

2.3　作用域

对于初学者而言，JavaScript的作用域有点让人困惑。其概念看起来很直观，但实际上并非如此。要想掌握作用域的概念，必须理解其中一些重要的细微之处。什么是作用域？在JavaScript中，作用域指的是代码当前的上下文。

一个变量的作用域就是该变量所在的上下文。作用域指明了可以从哪里访问到某个变量，以及在该上下文中是否可以访问这个变量。作用域分为全局作用域和局部作用域。

2.3.1　全局作用域

声明的所有变量默认都是定义在全局作用域中，这是JavaScript最恼人的设计决定之一。因为全局变量在所有的作用域中都是可见的，所以在任何作用域中都可以修改全局变量。全局变量导致很难在同一个程序/模块中运行松耦合的子程序。如果子程序碰巧也使用了同名的全局变量，就会造成干扰并有可能导致程序运行失败，而此类故障通常很难排查。这种现象有时被称为命名空间冲突（namespace clash）。上一章中讨论过全局作用域，现在来简单地回顾一下，搞明白如何以最佳方式避免此类问题。

可以采用两种方式创建全局变量。

- 第一种方法是在所有函数外部使用var语句。在函数外部声明的变量都属于全局作用域。
- 第二种方法是在声明变量的时候忽略var语句（也称为隐式全局变量）。我觉得这种设计是为了给新手提供方便，结果却是一场噩梦。即便是在函数内部，如果在声明变量的时候忽略了var，默认也是在全局作用域中创建变量。这就很不妙了。应该坚持对程序使用ESLint或JSHint，让它们帮你找出这种不当的地方。下面的例子展示了全局作用域的表现方式：

```
//全局作用域
var a = 1;
function scopeTest() {
  console.log(a);
}
scopeTest(); //打印出1
```

在本例中，我们在函数外部的全局作用域中声明了一个变量。在函数scopeTest()中可以使用该变量。如果在这个函数的作用域（局部作用域）中给全局变量赋一个新值，那么全局作用

域中该变量的原始值就会被覆盖：

```
//全局作用域
var a = 1;
function scopeTest() {
  a = 2; //忽略了var，全局变量被覆盖
  console.log(a);
}
console.log(a); //打印出1
scopeTest(); //打印出2
console.log(a); //打印出2（全局变量的值被重写）
```

2.3.2 局部作用域

和大多数编程语言不同，JavaScript没有块作用域（变量包含在大括号中），但是有函数作用域。在函数中声明的变量是局部变量，只能够在该函数中或是由该函数中的函数来访问：

```
var scope_name = "Global";
function showScopeName () {
  // 局部变量，只能在本函数中访问
  var scope_name = "Local";
  console.log (scope_name); // 打印出Local
}
console.log (scope_name); // 打印出Global
showScopeName(); // 打印出Local
```

2.3.3 函数作用域与块作用域

JavaScript变量以函数作为其作用域。可以将函数作用域想象成一个气泡，它使得气泡内部的变量不会被外部所看到。函数为在其内部声明的变量创建了一个这样的气泡。你可以像下面这样将气泡可视化：

```
-GLOBAL SCOPE---------------------------------------------|
var g =0;                                                 |
function foo(a) { -----------------------|                |
    var b = 1;                           |                |
    //code                               |                |
    function bar() { ------|             |                |
        // ...             |ScopeBar     | ScopeFoo       |
    }              ------|               |                |
    // code                              |                |
    var c = 2;                           |                |
}----------------------------------------|                |
foo(); //WORKS                                            |
bar(); //FAILS                                            |
----------------------------------------------------------|
```

JavaScript使用作用域链来建立某个函数的作用域。通常只有一个全局作用域，每个函数在其中有自己的嵌套作用域。在函数内部定义的函数也有局部作用域，该作用域与外围函数的作用域

是链接在一起的。**函数在作用域链中的位置与其出现在源代码中的位置是一致的**。在解析一个变量时，JavaScript从最内的作用域开始向外搜索。记住这一点，接着来看看JavaScript中的各种作用域规则。

在刚才那个粗糙的示意图中，可以看到函数foo()定义在全局作用域中。该函数具有自己的局部作用域，能够访问变量g，因为这个变量处于全局作用域中。变量a、b、c是在函数foo()的作用域中定义的，在该局部作用域中都可以使用。函数bar()也是一样。一旦超出了foo()的函数作用域，就无法再使用函数bar()了。在函数foo()（作用域气泡）的外部，既看不到也无法调用函数bar()。

函数bar()现在也有了自己的函数作用域(气泡),这里面都有些什么？它可以访问函数foo()以及在foo()的父作用域中创建的所有变量：a、b和c。它还可以访问全局变量g。

这是一个强有力的概念。花点时间思考一下。我们刚刚讨论过JavaScript中的全局作用域有多么肆无忌惮、难以控制。那么把一部分代码放进函数里怎么样？我们可以把这部分代码隐藏起来，并为其创建一个作用域气泡。利用函数建立作用域有助于编写出正确的代码，避免出现难以察觉的错误。

函数作用域以及在该作用域中隐藏变量和其他函数的另一个优势在于，能够避免标识符之间的冲突。下面的例子展示了这样的问题：

```
function foo() {
  function bar(a) {
    i = 2; // 在for循环的封闭作用域中修改i
    console.log(a+i);
  }
  for (var i=0; i<10; i++) {
    bar(i); // 死循环
  }
}
foo();
```

在bar()函数中，我们不小心修改了i的值。当在for循环中调用bar()时，变量i的值被设置成了2，因此再也跳不出这个无限循环了。这就是命名空间冲突的一个反面例子。

迄今为止，利用函数作为作用域在JavaScript中实现模块化和正确性，看起来是个不错的方法。嗯，尽管这种方法的确管用，但还算不上理想。第一个问题是必须得创建具名函数。如果创建这种函数仅仅是为了产生函数作用域，那么就会造成全局作用域或父作用域的污染。另外，我们还得不停地调用这些函数。由此所引入的大量的编写套路会使得代码的可读性越来越差：

```
var a = 1;
// 引入函数作用域
// 1. 在全局作用域中添加具名函数foo()
function foo() {
  var a = 2;
```

```
  console.log( a ); // 2
}
// 2. 调用具名函数foo()
foo();
console.log( a ); // 1
```

我们通过创建一个新函数foo()，在全局作用域中生成了函数作用域，并在随后调用该函数，执行相应的代码。

在JavaScript中，可以创建能够立即执行的函数来解决这些问题。仔细研究并输入下面的代码：

```
var a = 1;
// 引入函数作用域
// 1. 向全局作用域中添加具名函数foo()
(function foo() {
    var a = 2;
    console.log( a ); // 2
})(); //<---该函数立即执行
console.log( a ); // 1
```

注意，被包含在内的函数语句是以function开头的。这表明该函数并不是一个标准的函数声明，而是一个函数表达式。

作为表达式，(function foo(){ })语句意味着标识符foo只能在函数foo()的作用域中使用，无法用在外围作用域中。隐藏名称foo也不会对外围作用域造成不必要的污染。这样做的益处多多。我们可以在函数表达式之后加上()来立即执行它。完整的使用模式如下：

```
(function foo(){ /* 代码 */ })();
```

这种模式很常见，叫作IIFE（Immediately Invoked Function Expression，立即调用的函数表达式）。一些程序员在使用IIFE时会省去函数名。IIFE的主要用法是引入函数作用域，其实并不需要给函数命名。把之前的例子改写如下：

```
var a = 1;
(function() {
    var a = 2;
    console.log( a ); // 2
})();
console.log( a ); // 1
```

我们以IIFE的形式创建了一个匿名函数。尽管和之前的具名IIFE在效果上是一样的，但是匿名IIFE还是有一些缺点。

- 因为在栈跟踪（stack trace）过程中无法看到函数名，所以很难对这种代码进行调试。
- 无法对匿名函数使用递归（之前讨论过）。
- 过多地使用匿名IIFE有时会使得代码难以阅读。

Douglas Crockford和其他一些专家提出了一种形式略异的IIFE：

```
(function(){ /* 代码 */ }());
```

这两种IIFE形式都很常见，你会在很多代码中碰到。

IIFE也能接受参数。下面的例子展示了如何向IIFE传递参数：

```
(function foo(b) {
    var a = 2;
    console.log( a + b );
})(3); // 打印出5
```

2.3.4　行内函数表达式

行内函数表达式（inline function expression）还有另外一种流行的用法。在这种用法中，函数作为其他函数的参数：

```
function setActiveTab(activeTabHandler, tab){
  // 设置活动标签
  // 调用处理函数
  activeTabHandler();
}
setActiveTab( function (){
  console.log( "Setting active tab" );
}, 1 );
// 打印出Setting active tab
```

你也可以给行内函数表达式取一个名字，以便在调试代码时能够进行正确的栈跟踪。

2.3.5　块作用域

之前讲过，JavaScript并没有块作用域的概念。熟悉其他语言（如Java或C）的程序员会觉得很不适应。ECMAScript 6（ES6）引入了关键字`let`，可以用于生成传统的块作用域。如果你确定所使用的环境支持ES6，那么这就是一个极为方便的特性，你应该总是使用`let`。请看下面的代码：

```
var foo = true;
if (foo) {
  let bar = 42; // bar是该代码块的局部变量
  console.log( bar );
}
console.log( bar ); // ReferenceError
```

但就目前而言，大多数主流浏览器默认并不支持ES6。

到现在为止，你应该对JavaScript中作用域的工作方式有了比较清晰的理解。如果还是不太清楚，我建议你先停下来，重新看看本章前面几节内容。在互联网上搜索一下你不了解的地方，或是把问题发到Stack Overflow上。简而言之，一定要确保自己搞明白了作用域规则。

我们很自然地认为代码是从上到下一行接一行地执行的。大多数JavaScript代码也的确是这样执行的，但是也有一些例外。

考虑下面的代码：

```
console.log(a);
var a = 1;
```

如果你认为这是一段不合法的代码，在调用`console.log()`的时候会输出undefined，你完全正确。但如果是下面这样呢？

```
a = 1;
var a;
console.log( a );
```

上面这段代码会输出什么结果？鉴于语句`var a`出现在`a = 1`之后，那么变量`a`会被重新定义并赋以默认的undefined，输出结果自然应该是undefined，但结果其实是`1`。

JavaScript会把`var a = 1`划分成两个语句：`var a`和`a = 1`。第一个语句（也就是声明）是在编译阶段处理的，第二个语句（赋值）是在执行阶段处理的。

因此，之前的代码片段实际上是这样执行的：

```
var a; //----编译阶段

a = 1; //------执行阶段
console.log( a );
```

第一个代码片段是这样执行的：

```
var a; //-----编译阶段

console.log( a );
a = 1; //------执行阶段
```

如你所见，变量和函数声明在编译阶段被移到了代码的顶部——这就是常说的**提升**（hoisting）。一定要记住，只有声明才会被提升，而赋值或其他可执行的逻辑依然保留在原位置。下面的代码片段展示了函数声明提升：

```
foo();
function foo() {
  console.log(a); // undefined
  var a = 1;
}
```

因为函数`foo()`的声明会被提升，所以我们可以在其定义之前就调用该函数。关于提升很重要的一点是，它是以作用域为单位进行的。在函数`foo()`中，变量声明会被提升到函数内部的顶部，而不是整个程序的顶部。提升后的函数`foo()`会变成下面的样子：

```
function foo() {
  var a;
  console.log(a); // undefined
  a = 1;
}
```

函数表达式并不会像函数声明那样被提升，下面来讲讲为什么会这样。

2.4　函数声明与函数表达式

定义函数有两种方法。尽管作用一样，但这两种形式之间存在差异。来看下面的例子：

```
// 函数表达式
functionOne();
// 错误
// TypeError: functionOne is not a function

var functionOne = function() {
  console.log("functionOne");
};
// 函数声明
functionTwo();
// 正确
// 打印出functionTwo

function functionTwo() {
  console.log("functionTwo");
}
```

函数声明是在执行流程进入到该函数所处的上下文时处理的，这个处理过程在代码实际执行前进行。对于带有函数名的函数（在上面的例子中是`functionTwo()`），其名称会被保存在函数声明所在的作用域中。该过程是在作用域中代码被执行前完成的，因此在定义前调用`functioTwo()`不会产生错误。

但`funtionOne()`是一个匿名函数表达式，它是在代码执行到其所在位置时进行求值的（也称为运行期执行），必须在调用前声明。

所以说，`functionTwo()`的函数声明会被提升，而`functionOne()`的函数表达式会在执行流逐行到达其位置时执行。

> 函数声明和变量声明都会被提升，但是函数在先，变量在后。

要记住的是，绝不要根据条件来使用函数声明。这种行为是非标准化的，在不同平台上的结果并不一致。下面的例子中就展示了这种用法，依据不同条件进行函数声明。我们尝试给函数`sayMoo()`赋以不同的函数体，但是这种条件代码并不能保证在所有的浏览器上都能正常工作，

有可能会产生无法预测的结果：

```
// 千万别这么做——不同的浏览器行为也不同
if (true) {
  function sayMoo() {
    return 'trueMoo';
  }
}
else {
  function sayMoo() {
    return 'falseMoo';
  }
}
foo();
```

其实，完全可以使用函数表达式巧妙地实现同样的效果：

```
var sayMoo;
if (true) {
  sayMoo = function() {
    return 'trueMoo';
  };
}
else {
  sayMoo = function() {
    return 'falseMoo';
  };
}
foo();
```

如果你好奇为什么不能在条件语句块中使用函数声明，请继续往下读；否则，可以跳过下面一段。

函数声明只允许出现在程序或者函数体中，不能出现在块结构（`{...}`）中。块只能包含语句，不能包含函数声明。因为这个原因，几乎所有的JavaScript实现在这一点上的行为都各不相同。最好不要在条件语句块中使用函数声明。

函数表达式非常流行。在JavaScript程序员中很常见的一种用法是根据某种条件生成函数定义。生成的函数通常都是出现在同一个作用域中，因此必须使用函数表达式。

2.5 arguments 参数

arguments参数是传递给函数的所有参数的集合。这个集合有一个名为`length`的属性，它包含了参数的个数，单个的参数值可以使用数组的索引记法来获得。好吧，我们说的其实有一点不对。arguments参数并不是JavaScript数组，如果你试图在arguments上使用数组方法，肯定会失败。可以把arguments视为一个类似数组的结构，它能够帮助我们编写可以接受可变数量参数的函数。在下面的代码片段中，你可以给函数传入数量不一的参数，并使用arguments对其进行迭代：

```
var sum = function () {
  var i, total = 0;
  for (i = 0; i < arguments.length; i += 1) {
    total += arguments[i];
  }
  return total;
};
console.log(sum(1,2,3,4,5,6,7,8,9)); // 打印出45
console.log(sum(1,2,3,4,5)); // 打印出15
```

之前我们讨论过，arguments参数并不是真正的数组，可以按照下面的方法将其转换成数组：

```
var args = Array.prototype.slice.call(arguments);
```

转换成数组之后，就可以按照需要进行操作了。

this 参数

当函数被调用时，除了在调用时明确给出的那些参数，还有一个叫作this的隐式参数也会被传入函数。它指向一个与此次函数调用相关联的对象，该对象被称为**函数上下文**（function context）。如果你写过Java代码，应该不会对this关键字感到陌生；与Java类似，this指向定义了方法的类的实例。

了解这些知识后，让我们来看看各种调用方式。

1. 作为函数调用

如果函数不是以方法、构造函数或通过apply()、call()调用，那么它只是**作为一个函数**被调用：

```
function add() {}
add();
var substract = function() {

};
substract();
```

当函数以这种模式调用时，this被绑定在全局对象上。很多专家认为这是一个糟糕的设计选择。照理说，this应该被绑定在父上下文（parent context）上。在这种情况下，你可以把this的值保存在另一个变量中。随后我们会专门讲述这种调用模式。

2. 作为方法调用

方法是作为对象属性的函数。对于方法来说，this被绑定在方法被调用时所在的对象上：

```
var person = {
  name: 'Albert Einstein',
  age: 66,
```

```
  greet: function () {
    console.log(this.name);
  }
};
person.greet();
```

在本例中，this被绑定在greet被调用时所在的person对象上，因为greet是person的一个方法。让我们来对比一下在这两种调用模式中this的变化。

来看下面的HTML和JavaScript代码：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
    function testF(){ return this; }
    console.log(testF());
    var testFCopy = testF;
    console.log(testFCopy());
    var testObj = {
      testObjFunc: testF
    };
    console.log(testObj.testObjFunc ());
  </script>
</head>
<body>
</body>
</html>
```

在Firebug控制台中，可以看到以下输出：



前两次调用都是函数调用。因此，this的值指向的是全局上下文（在这里是Window）。

接下来，我们定义了一个名为testObj的对象，该对象包含一个testObjFunc属性，属性值是testF()的引用——如果你没有意识到生成了对象，也不用担心。这样一来，我们就创建了testObjFunc()方法。如果调用该方法，在显示出this的值的时候就能够看出函数的上下文了。

3. 作为构造函数调用

构造函数的声明和其他函数一样，作为构造函数使用的函数也没有什么特别之处。但是构造

函数的调用方法却大相径庭。

要以构造函数的方式调用函数，需要在函数调用前加上关键字new。这样的话，this就被绑定在新创建的对象上了。

在进一步讨论之前，先简要介绍一下JavaScript中的面向对象。当然，我们会在下一章中详细讲述这个话题。JavaScript是一门原型继承的语言（prototypal inheritance language），也就是说对象可以直接从其他对象中继承属性。JavaScript中没有类。使用new前缀调用的函数被称为构造函数。通常为了易于区分，构造函数并没有采用**驼峰拼写法**（CamelCase），而是采用了**Pascal拼写法**（PascalCase）。[①]注意在下面的例子中，函数greet使用this来访问name属性，this参数被绑定在Person对象上：

```
var Person = function (name) {
  this.name = name;
};
Person.prototype.greet = function () {
  return this.name;
};
var albert = new Person('Albert Einstein');
console.log(albert.greet());
```

在下一章学习对象的时候，我们来讨论这种调用方法。

4. 通过apply()和call()方法调用

我们之前说过JavaScript函数也是对象。和其他对象一样，函数也具备方法。要想使用apply()方法调用函数，需要传入两个参数：作为函数上下文的对象以及作为调用参数的数组。call()方法的用法也差不多，除了调用参数不能作为数组传入，而是要直接写成参数列表的形式传入。

2.6　匿名函数

本章前面介绍过匿名函数，考虑到这是一个非常重要的概念，我们将对其作进一步的讲解。由于借鉴了Scheme，匿名函数是JavaScript中重要的逻辑与结构构件。

匿名函数通常用于无需函数名的情况。下面来看一些匿名函数最常见的用法。

2.6.1　对象创建过程中的匿名函数

匿名函数可以作为对象的属性。这样，就可以使用点操作符（.）调用该函数了。如果你有Java或其他面向对象语言的背景，对这种用法绝对不会陌生。在这类语言中，作为类的组成部分的函数，通常采用Class.function()的写法来调用。来看下面的例子：

① Pascal拼写法中第一个单词的首字母大写，而驼峰拼写法中第一个单词的首字母小写。——译者注

```
var santa = {
  say :function(){
    console.log("ho ho ho");
  }
}
santa.say();
```

上例中创建了一个包含say属性的对象，该属性值是一个匿名函数。在这种情况下，这个属性不再叫作函数，而是被称为方法。不需要函数名的原因在于我们打算将其作为对象属性来调用，这是一种常用的模式，用起来很方便。

2.6.2　列表创建过程中的匿名函数

在这里，我们创建了两个匿名函数，并将它们添加到了一个数组里（随后会详细讨论数组）。然后遍历该数组，在每次循环过程中执行对应的函数：

```
<script type="text/javascript">
var things = [
  function() { alert("ThingOne") },
  function() { alert("ThingTwo") },
];
for(var x=0; x<things.length; x++) {
  things[x]();
}
</script>
```

2.6.3　作为函数参数的匿名函数

这是最常用的模式之一，在大多数专业代码库中都可以看到此类用法：

```
// 函数语句
function eventHandler(event){
  event();
}

eventHandler(function(){
  // 执行事件相关的操作
  console.log("Event fired");
});
```

在上面的代码中，我们将匿名函数传给了另一个函数。在接收函数中，执行作为参数传入的函数。如果要创建的是一次性（single-use）函数，比如对象方法或事件处理程序，这种用法就很方便了。匿名函数的语法要比两步走的函数声明（先声明函数，然后描述函数操作）更紧凑。

2.6.4　出现在条件逻辑中的匿名函数

可以使用匿名函数表达式来根据条件改变行为。下面的例子展示了这种使用模式：

```
var shape;
if(shape_name === "SQUARE") {
  shape = function() {
    return "drawing square";
  }
}
else {
  shape = function() {
    return "drawing square";
  }
}
alert(shape());
```

在这里，根据不同的条件为shape变量赋以不同的函数。善加利用的话，这种模式大有用处。如果不加节制，则会产生难以阅读及排错的代码。

在本书随后的部分中，我们还会看到一些函数式编程技巧，例如**记忆化**（memoization）以及函数调用缓存（caching functions call）。如果本章前面的内容你都是快速浏览的，建议你现在先停下来，思考一下之前所讨论过的内容。接下来的几页中包含了大量的信息，需要花些工夫才能够理解掌握。最好在继续往下读之前把本章再重读一遍。下一节重点讲闭包和模块。

2.7　闭包

闭包是纯函数式编程语言的传统特性之一。通过将闭包视为核心语言构件的组成部分，JavaScript语言展示了其与函数式编程语言的紧密联系。由于能够简化复杂的操作，闭包在主流JavaScript库以及高水平产品代码中日益流行起来。你会发现，有经验的JavaScript程序员在谈到闭包的时候几乎都是毕恭毕敬的，好像这东西是某种超出了凡人理解能力的神器一般。其实并非如此。在学习闭包的概念时，你会发现其非常直观易懂，几乎都是明摆着的。在领会闭包之前，建议你通过反复阅读本章内容、在互联网上查找相关资料、编写代码、阅读JavaScript库来理解闭包的工作原理——别轻言放弃。

首先要认识到的就是：闭包在JavaScript中是无所不在的。它并不是语言的某种隐藏特性。

在寻根求源之前，来快速回顾一下JavaScript的词法作用域（lexcial scope）。我们详细讨论过，JavaScript中如何在函数一级上确定词法作用域。词法作用域实际上决定了所有标识符的声明位置和方式，预测了标识符在执行过程中的查找过程。

简单地说，闭包是在函数声明时所创建的作用域，它使得函数能够访问并处理函数的外部变量。换句话说，闭包可以让函数访问到在函数声明时处于作用域中的所有变量以及其他函数。

来看几个样例代码，理解一下这个定义：

```
var outer = 'I am outer'; // 在全局作用域中定义一个变量
function outerFn() { // 在全局作用域中声明一个函数
```

```
  console.log(outer);
}
outerFn(); // 打印出I am outer
```

等着看一些不一般的结果？没有的，这其实都是闭包最普通的用法。我们在全局作用域中分别声明了一个变量和一个函数。在函数中，能够访问到全局作用域中声明的变量outer。所以实质上，函数outerFn()的外围作用域就是一个闭包，总是可为其所用。这是个不错的开始，那么你可能会纳闷为什么这样就能说闭包是个好东西了？

来个复杂点的例子：

```
var outer = 'Outer'; // 全局变量
var copy;
function outerFn(){ // 全局函数

  var inner = 'Inner'; // 该变量只有函数作用域，无法从外部访问

  function innerFn(){ // outerFn()中的innerFn()
    // 全局上下文和外围上下文都可以在这里使用，
    // 因此可以访问到outer和inner
    console.log(outer);
    console.log(inner);
  }
  copy=innerFn; // 保存innerFn()的引用
  // 因为copy是在全局上下文中声明的，所以在外部可以使用
}
outerFn();
copy(); // 不能直接调用innerFn()，但是可以通过在全局作用域中声明的变量来调用
```

来分析一下上面的例子。在innerFn()中可以访问变量outer，因为它处于全局上下文中。在执行完outerFn()之后，执行了innerFn()，这是通过将该函数的引用复制到一个全局变量copy中来实现的。在利用变量copy调用函数innerFn()执行时，此刻已经不在outerFn()的作用域中了。因此下面的代码不是应该失败吗？

```
console.log(inner);
```

变量inner的值应该是undefined吧？可是，上面代码片段的输出却是：

```
"Outer"
"Inner"
```

究竟是什么使得函数innerFn()在执行时仍然能够访问变量inner，即便是在创建该变量的作用域已经不存在很久之后？当我们在outerFn()中声明innerFn()时，不仅定义了函数，还创建了一个闭包，其中包含了所声明的函数以及在声明时处于作用域中的所有变量。在执行innerFn()时，尽管声明该函数的作用域已经不在了，但是它仍能够通过闭包访问当初声明时所在的作用域。

把这个例子做一个扩展，来看看闭包还能做些什么：

```
var outer='outer';
var copy;
function outerFn() {
  var inner='inner';
  function innerFn(param){
    console.log(outer);
    console.log(inner);
    console.log(param);
    console.log(magic);
  }
  copy=innerFn;
}
console.log(magic); //ERROR: magic not defined
var magic="Magic";
outerFn();
copy("copy");
```

在上面的例子中，我们添加了一些新内容。首先，向innerFn()添加一个参数——这仅仅是为了演示参数也是闭包的一部分。我们要重点强调两点。

外围作用域中的所有变量都包含在闭包中，哪怕它们的声明出现在函数声明之后。这正是innerFn()中的console.log(magic)能够正常工作的原因。

但是在全局作用域中的console.log(magic)却出现了错误，这是因为就算在相同的作用域中，未定义的变量也不能被引用。

所有这些例子都旨在传达一些与闭包工作原理息息相关的概念。闭包是JavaScript语言中的重要特性，在大多数库中都能看到它。

让我们来看几个常见的闭包使用模式。

2.8　计时器和回调函数

在实现计时器或回调函数时，大多需要在之后的某个时间点上异步调用处理程序。由于是异步调用，我们需要在相关函数中访问作用域外的变量。考虑下面的例子：

```
function delay(message) {
  setTimeout( function timerFn(){
    console.log( message );
  }, 1000 );
}
delay( "Hello World" );
```

我们将内部的函数timerFn()传递给内建的库函数setTimeout()。但是timerFn()有一个覆盖了delay()作用域的作用域闭包，因此它能够访问到变量message。

2.9 私有变量

闭包常用来将信息封装成私有变量的形式。在JavaScript中无法使用Java或C++中的类似封装方法，但通过利用闭包，我们也可以实现类似的效果：


```
function privateTest(){
  var points=0;
  this.getPoints=function(){
    return points;
  };
  this.score=function(){
    points++;
  };
}

var private = new privateTest();
private.score();
console.log(private.points); // undefined
console.log(private.getPoints());
```

在上例中，我们创建了一个函数，打算将其作为构造函数调用。在`privateTest()`函数中，创建了函数作用域变量`points`。这个变量只能在`privateTest()`中使用。另外，还创建了一个访问器（accessor）函数（也称为接收器，getter）`getPoints()`，这个方法允许从`privateTest()`外部只读取变量points的值，使得该变量成为函数的私有变量。而另一个方法`score()`允许在不直接从外部访问的情况下修改私有变量points的值。这样，我们写出的代码就能够以一种受控的方式来更新私有变量。在编写库的时候，如果你希望根据约定和预先建立的接口来控制变量访问方式的话，这种模式非常有用。

2.10 循环与闭包

考虑下面这段在循环中使用函数的代码：

```
for (var i=1; i<=5; i++) {
  setTimeout( function delay(){
    console.log( i );
  }, i*100);
}
```

这应该会在控制台中以100ms的间隔分别打印出1、2、3、4、5，对不对？但实际的结果是以100ms为间隔，分别打印出6、6、6、6、6。这是怎么回事？在这里，碰到的是一个与闭包和循环有关的常见问题。变量`i`的值在函数被绑定后更新，这就意味着每一个被绑定的函数打印出的总是变量`i`中保存的最后那个值。实际上，`setTimeout`的回调函数是在循环结束后才执行的。如果你试图以这种方式在循环中使用函数，JSLint会发出警告信息。

怎么修改这个错误呢？我们可以引入一个函数作用域，并在该作用域中建立变量i的本地副本。下面的代码片段就展示了正确的做法：

```
for (var i=1; i<=5; i++) {
  (function(j){
    setTimeout( function delay(){
      console.log( j );
    }, j*100);
  })( i );
}
```

将变量i作为参数传入，并将其复制到IIFE的局部变量j中。IIFE会针对每次迭代创建一新的作用域，使用正确的值来更新局部变量。

2.11　模块

模块可用于模拟类，强调的是对变量和函数的公共及私有访问。模块有助于减少全局作用域污染。有效地使用模块能够降低大型代码基础库之间的名称冲突。这种模式的典型用法如下：

```
var moduleName=function() {
  // 私有状态
  // 私有函数
  return {
    // 公共状态
    // 公共变量
  }
}
```

要实现以上模式有两个要求。

- 必须有一个外围函数，至少需要执行一次。
- 外围函数必须返回至少一个内部函数（inner function）。这需要创建一个涵盖了私有状态的闭包，否则无法访问私有状态。

请看下面的模块代码：

```
var superModule = (function (){
  var secret = 'supersecretkey';
  var passcode = 'nuke';

  function getSecret() {
    console.log( secret );
  }

  function getPassCode() {
    console.log( passcode );
  }

  return {
```

```
    getSecret: getSecret,
    getPassCode: getPassCode
  };
})();
superModule.getSecret();
superModule.getPassCode();
```

这个例子满足了上面两个条件。首先，我们创建了一个IIFE（具名函数也可以）作为外围（outer enclosure）。定义在其中的变量将会保持私有状态，因为它们的作用域被限制在函数内部。我们将公共函数返回，以确保拥有一个覆盖了私有作用域的闭包。在模块模式中使用IIFE实际上会产生该函数的一个单例。如果你想生成多个实例，需要创建具名函数表达式，将其作为模块的一部分。

我们还会继续学习JavaScript中和函数相关的方方面面以及闭包。有很多与此相关的新颖用法。理解各种模式的一种有效方法就是研究流行的JavaScript库的代码，在自己的代码中实践这些模式。

编码风格

和上一章一样，我们来考虑一些编码风格并以此作结。再次重申，这些风格只是作为一种指南，并非规则——如果你认为有其他更好的方案，只管采用就行了。

- 使用函数声明，不使用函数表达式：

```
// 不建议
const foo = function () {
};

// 建议
function foo() {
}
```

- 绝不要在非函数块（if、while等）中声明函数。可以选择将函数赋给变量。尽管浏览器允许你这么做，但是不同的浏览器对此的解释方式各不相同。
- 不要给参数起`arguments`的名字，这会使其在函数作用域中的优先级高于`arguments`对象。

2.12 小结

在本章中，我们学习了JavaScript函数。在JavaScript中，函数扮演着重要的角色。我们讨论了函数的创建和使用、闭包的重要概念以及函数中的变量作用域，另外还研究了将函数作为创建可见类和封装的方法。

在下一章中，我们将学习JavaScript中的各种数据结构以及数据处理技术。

第3章 数据结构及相关操作

在大多数的编程时间里，你都免不了要跟数据打交道。这包括处理数据属性、根据数据进行推演以及修改数据性质。在本章中，我们将详细地学习JavaScript中的各种数据结构以及数据处理技术。学会这些语言构件的正确用法，才能够写出准确、简洁、易读、更快的程序。本章包含以下主题：

- 正则表达式
- 严格匹配
- 匹配字符组
- 重复出现
- 首部与尾部
- 向后引用
- 贪婪限定符以及惰性限定符
- 数组
- map
- set
- 编码风格

3.1 正则表达式

如果你还不熟悉正则表达式，最好花时间学习一下。学习有效地使用正则表达式是回报最高的技能之一。在大多数的代码评审过程中，我发表的第一条意见就是如何把一段代码转化成一行**正则表达式**（regular expression，RegEx）。如果你研究过流行的JavaScript库，就会惊讶于正则表达式的无所不在。大多数有经验的工程师都离不开它，这主要是因为一旦学会，正则表达式用起来既精炼又易于测试。不过，学习正则表达式可是需要下一番功夫。正则表达式是一种表示匹配字符串的模式的方法。表达式本身是由项（term）和操作符（operator）组成的，它们让我们得以定义这些模式。稍后会介绍这些项和操作符的组成。

在JavaScript中，创建正则表达式的方法有两种：通过正则表达式字面量和构造RegExp对象的实例。

举例来说，如果你想创建一个能够严格匹配字符串test的正则表达式，可以使用下面的正则表达式字面量：

```
var pattern = /test/;
```

正则表达式使用斜杠作为分隔符。我们也可以将正则表达式作为字符串传入RegExp()，从而创建出一个RegExp实例：

```
var pattern = new RegExp("test");
```


这两种格式会生成相同的正则表达式，并保存在变量pattern中。除了正则表达式本身之外，还有三个标志可以配合使用。

- i：可以使正则表达式忽略大小写，这样的话，/test/i不仅能匹配test，还能匹配Test、TEST、tEsT等。
- g：相较于只匹配模式的第一次出现，g标志可以使正则表达式匹配模式的所有实例。随后会有详细说明。
- m：可以使正则表达式跨多行（例如textarea元素的值）进行匹配。

这些标志要放在正则表达式字面量的尾部（例如/test/ig），或是作为第二个字符串参数传给RegExp构造函数（new RegExp("test", "ig")）。

下面的例子演示了各种标志的用法，以及它们是如何影响模式匹配的：

```
var pattern = /orange/;
console.log(pattern.test("orange")); // true

var patternIgnoreCase = /orange/i;
console.log(patternIgnoreCase.test("Orange")); // true

var patternGlobal = /orange/ig;
console.log(patternGlobal.test("Orange Juice")); // true
```

如果只是测试模式是否能匹配某个字符串，那就没太大的意思了。让我们来看看如何表示更复杂的模式。

3.2　严格匹配

任何非正则表达式字符或操作符的字符序列，代表的都是该字符本身：

```
var parttern = /orange/;
```

这里要表示的是o后面跟着r，r后面跟着a，a后面跟着n……明白了吧。在使用正则表达式的时候，我们很少采用严格匹配，因为这和直接比较两个字符串没什么分别。严格匹配有时候也叫作简化模式（simple pattern）。

3.3　匹配字符组

如果想针对一组字符进行匹配，可以将这组字符放入[]。例如，[abc]表示a、b或c中任意一个字符：

```
var pattern = /[abc]/;
console.log(pattern.test('a')); //true
console.log(pattern.test('d')); //false
```

也可以在模式开头加上一个^（脱字符）来指定不想匹配到的字符：

```
var pattern = /[^abc]/;
console.log(pattern.test('a')); //false
console.log(pattern.test('d')); //true
```

这种模式还有另一种很重要的用法是用来指明值的范围。如果想匹配字符或数字的某个连续范围，可以使用下面的模式：

```
var pattern = /[0-5]/;
console.log(pattern.test(3)); //true
console.log(pattern.test(12345)); //true
console.log(pattern.test(9)); //false
console.log(pattern.test(6789)); //false
console.log(/[0123456789]/.test("This is year 2015")); //true
```

像$和.这样的特殊字符，要么代表匹配除自身之外的其他字符，要么作为限定其之前项的操作符。实际上，我们已经看到了字符[、]、-和^可用来表示各自字面值之外的含义。

我们如何匹配[、$、^或其他特殊字符的字面量呢？在正则表达式中，反斜杠字符可以用来对其后的字符进行转义，从而实现字面量的匹配。因此，\[匹配的就是一个普通的字符[，而不是字符组表达式的开括号。双反斜杠（\\）可以匹配一个普通的反斜杠。

上例中，我们看到test()方法可以根据模式是否匹配来返回true或false。但有时候需要访问特定模式所出现的位置，而exec()方法可以解决这个问题。

exec()方法接受单个字符串作为参数，返回一个包含了所有匹配的数组。考虑下面的例子：

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/;
var arrMatches = regExAt.exec(strToMatch);
console.log(arrMatches);
```

这个代码片段的输出是['Toy']。如果你需要模式Toy所有的匹配实例，可以使用g（global）标志：

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/g;
var arrMatches = regExAt.exec(strToMatch);
console.log(arrMatches);
```

这将返回原始文本中所出现的所有Toy。String对象有一个match()方法，其功能与exec()方法类似。match()需要在String对象上调用，并将正则表达式作为参数传入。考虑下面的例子：

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/;
var arrMatches = strToMatch.match(regExAt);
console.log(arrMatches);
```


在本例中，我们在String对象上调用了match()方法，并将正则表达式作为参数一并传入。可以看到两种方法的结果是一样的。

还有一个String对象的方法是replace()，它可以使用其他字符串来替换某个子串：

```
var strToMatch = 'Blue is your favorite color ?';
var regExAt = /Blue/;
console.log(strToMatch.replace(regExAt, "Red"));
// 输出"Red is your favorite color ?"
```

可以将函数作为replace()方法的第二个参数。该函数使用匹配的文本作为参数，将用来替换的文本作为函数值返回：

```
var strToMatch = 'Blue is your favorite color ?';
var regExAt = /Blue/;
console.log(strToMatch.replace(regExAt, function(matchingText){
  return 'Red';
}));
// 输出"Red is your favorite color ?"
```

String对象的split()方法也可以接受正则表达式作为参数并返回一个数组，该数组中包含了经过分割后的所有子串：

```
var sColor = 'sun,moon,stars';
var reComma = /\,/;
console.log(sColor.split(reComma));
// 输出["sun", "moon", "stars"]
```

因为逗号在正则表达式中有特殊含义，所以如果需要使用它的字面意义的话，需要在它前面加上一个反斜杠进行转义。

使用简单字符组就可以匹配多个模式。假如想匹配cat、bat和fat，下面的代码片段展示了具体的做法：

```
var strToMatch = 'wooden bat, smelly Cat,a fat cat';
var re = /[bcf]at/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["bat", "Cat", "fat", "cat"]
```

如你所见，这种用法可以写出更简洁的正则表达式。来看下面的例子：

```
var strToMatch = 'i1,i2,i3,i4,i5,i6,i7,i8,i9';
var re = /i[0-5]/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["i1", "i2", "i3", "i4", "i5"]
```

在本例中，我们使用范围`[0-5]`来匹配待匹配字符串中的数字部分，得到的匹配结果是从`i0`到`i5`。你也可以使用否定字符^来过滤出其余的匹配：

```
var strToMatch = 'i1,i2,i3,i4,i5,i6,i7,i8,i9';
var re = /i[^0-5]/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["i6", "i7", "i8", "i9"]
```

注意，我们否定的只是范围子句，而非整个表达式。

有些字符组有对应的快捷写法。例如，`\d`表示的意义和`[0-9]`相同。

写　法	含　义
`\d`	任意的单个数字字符
`\w`	任意单个字母或数字字符（单词字符）
`\s`	任意的单个空白字符（空格、制表符、换行符等）
`\D`	任意的单个非数字字符
`\W`	任意的单个非字母或数字字符
`\S`	任意的单个非空白字符
`.`	除换行符之外的任意单个字符

这些快捷写法是书写简洁的正则表达式的关键。看下面的例子：

```
var strToMatch = '123-456-7890';
var re = /[0-9][0-9][0-9]-[0-9][0-9][0-9]/;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["123-456"]
```

这个表达式看起来有点奇怪。我们可以使用`\d`来替换`[0-9]`，让它变得更易读：

```
var strToMatch = '123-456-7890';
var re = /\d\d\d-\d\d\d/;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
```

```
//["123-456"]
```

不过，你很快会发现还有更好的方法可以实现类似的效果。

3.4 重复出现

迄今为止，我们已经学习了如何匹配固定的字符或数字模式。但大多数时候，也需要处理某些具有重复性的模式。举例来说，如果想匹配4个a，那么可以写作/aaaa/，但如果要指定一个能够匹配任意数量a的模式，该怎么办呢？

正则表达式提供了各式各样的重复限定符。重复限定符可以用来指定某个特定的模式能够出现多少次。我们可以指定一个固定的值（字符应该出现*n*次），也可以指定可变的值（字符至少出现*n*次，至多出现*m*次）。各种重复限定符如下。

- ?：出现0次或1次（将模式视为可选的）
- *：出现0次或多次
- +：出现1次或多次
- {n}：只出现n次
- {n，m}：出现n到m次
- {n，}：至少出现n次
- {，n}：出现0到n次

在下面的例子中，我们创建了一个模式，其中字符u是可选的（出现0或1次）：

```
var str = /behaviou?r/;
console.log(str.test("behaviour"));
// true
console.log(str.test("behavior"));
// true
```

可以将表达式/behaviou?r/读作“字符u出现0或1次”，这样有助于理解。重复限定符要紧跟在希望出现重复的字符后面。再来看一个例子：

```
console.log(/'\d+'/.test("'123'")); // true
```

你可以将表达式'\d+'理解为：'用来匹配普通字符'，\d用来匹配字符[0-9]，限定符+允许重复一次或多次，'用来在此匹配普通字符'。

也可以使用()将字符表达式分组。观察下面的例子：

```
var heartyLaugh = /Ha+(Ha+)+/i;
console.log(heartyLaugh.test("HaHaHaHaHaHaHaaaaaaaaaaa"));
//true
```

让我们将上面的表达式进行分解来理解其用途。

- H：匹配字符的字面意义
- a+：字符a出现1次或多次
- (：表达式分组开始
- H：匹配字符的字面意义
- a+：字符a出现1次或多次
-)：表达式分组结束
- +：表达式分组（Ha+）出现1次或多次

现在，分组的用法应该更清晰了。如果需要解释这个正则表达式，像前面的例子中那样拆开分析会有助于理解。

经常需要将一系列字母或数字作为整体而非子串进行匹配。这种做法相当频繁地出现在需要匹配独立单词（一个完整的单词，而不是作为其他单词的组成部分）的时候。可以使用\b模式来指定单词边界。\b匹配的是这样一个位置：一边是单词字符（字母、数字或下划线），另一边是非单词字符。

下面的例子是一个简单的字面匹配。如果cat是一个子串的话，也能够成功匹配：

```
console.log(/cat/.test('a black cat')); //true
```

但是在下面的例子中，我们在单词cat之前使用\b定义了一个单词边界，也就是说待匹配的cat必须是一个完整的单词，而非子串。单词边界出现在cat之前，因此在文本a black cat中能够找到相应的匹配：

```
console.log(/\bcat/.test('a black cat')); //true
```

如果我们将相同的模式用于单词tomcat，这次就没法匹配了，因为单词tomcat中的cat前面并没有单词边界：

```
console.log(/\bcat/.test('tomcat')); //false
```

单词tomcat中的cat后面有一个单词边界，因此下面的测试能够成功匹配：

```
console.log(/cat\b/.test('tomcat')); //true
```

在下面的例子中，我们在单词cat前后指定了单词边界，以此表明cat应该是一个前后都有边界的独立单词：

```
console.log(/\bcat\b/.test('a black cat')); //true
```

基于同样的逻辑，下面的匹配肯定会失败，因为单词concatenate中cat的前后并没有边界：

```
console.log(/\bcat\b/.test("concatenate")); //false
```

exec()方法在获取匹配信息方面很有用，因为它会返回一个包含匹配信息的对象。exec()返回的对象有一个index属性，可以告诉我们成功匹配出现在字符串中的哪个位置。这个功能在不少地方都能派上用场：

```
var match = /\d+/.exec("There are 100 ways to do this");
console.log(match);
// ["100"]
console.log(match.index);
// 10
```

选择结构

选择结构可以使用|（管道符）来表示。例如，/a|b/可以匹配字符a或b，/(ab)+|(cd)+/可以匹配一个或多个ab或cd。

3.5 首部与尾部

我们经常需要确保模式在字符串的首部或尾部进行匹配。当脱字符（^）用作正则表达式的第一个字符的时候，可以将匹配过程锁定在字符串的开头，因此，/^test/只能够匹配出现在待匹配字符串起始位置上的test子串。与此类似，美元符号（$）表示模式必须出现在字符串的尾部：/test$/。

^和$配合使用，表明指定的模式必须涵盖整个待匹配的字符串：/^test$/。

3.6 向后引用

表达式完成求值之后，每一个分组所匹配的内容都会被保存起来以备后用。这些值被称为向后引用（backreference）。向后引用是按照从左到右的方向，依据开括号的先后次序来创建并编号的。你可以将向后引用视为字符串中已经成功匹配的部分，这些部分对应着正则表达式中的项（term）。

向后引用可以写作一个反斜杠后面跟着一个从1开始的编号，用于引用已捕获（匹配）的内容，比如\1、\2等。

比如说/^([XYZ])a\1/，它能够匹配这样一个字符串：以X、Y、Z中任意字符开头，然后是字符a，接着是第一个分组中捕获的内容。这和/[XYZ] a[XYZ]/大不相同。在前者中，a后面的字符不能是任意的X、Y、Z，而必须与之前所匹配的第一个字符相同。向后引用同String对象的replace()方法配合使用的时候，需要用到特殊的字符序列$1、$2等。假如你想将字符串1234 5678更改成5678 1234，可以使用下面的代码：

```
var orig = "1234 5678";
var re = /(\d{4}) (\d{4})/;
var modifiedStr = orig.replace(re, "$2 $1");
console.log(modifiedStr); // 输出"5678 1234"
```

本例中的正则表达式有两个分组，每组包含4个数字。在replace()方法的第二个参数中，$2等于5678，$1等于1234，和其出现在模式中的次序相对应。

3.7　贪婪限定符与惰性限定符

到目前为止，我们所讨论的所有限定符都是贪婪性质的。贪婪限定符针对整个字符串进行匹配。如果没有发现匹配，它会删除字符串的最后一个字符，然后再尝试匹配。如果还没匹配，再删除最后一个字符，这个过程一直重复，直到出现匹配或是字符串已经变成空串。

例如，模式\d+能够匹配一个或多个数字。如果字符串是123的话，贪婪匹配可以匹配到1、12和123。贪婪模式h.+l可以匹配字符串hello中的hell——这是能够匹配的最长的字符串。因为\d+是贪婪匹配，所以它会尽可能多地匹配数字，故最后的匹配结果就是123。

与贪婪限定符相反，惰性限定符则是尽可能少地匹配字符。可以在正则表达式后面加上问号（?），使其成为惰性匹配。惰性模式h.?l可以匹配字符串hello中的hel—— 这是能够匹配到的最短的字符串。

模式\w*?X可以匹配到0个或多个单词以及一个X。但是*后的?表示应该尽可能少地匹配字符。对于字符串abcXXX，匹配结果可以是abcX、abcXX或abcXXX，那究竟应该匹配哪一个呢？因为*?是惰性模式，所以应该尽可能少地匹配，因此最后的匹配结果是abcX。

有了这些必备知识，让我们来尝试利用正则表达式解决一些常见的问题。

删除字符串首尾多余的空白字符是一个极其常见的用法。直到最近，String对象本身都没有trim()方法，一些JavaScript库为没有String.trim()方法的旧浏览器提供了字符串修剪功能。最常用的方法如下所示：

```
function trim(str) {
  return (str || "").replace(/^\s+|\s+$/g, "");
}
 console.log("--"+trim("   test    ")+"--");
//"--test--"
```

如果我们想把重复的空白字符替换成单个呢？

```
re=/\s+/g;
console.log('There are     a lot        of spaces'.replace(re,' '));
//"There are a lot of spaces"
```

在上面的代码片段中，我们尝试匹配一个或多个空格字符序列，然后将其替换成单个空格。

如你所见，正则表达式就像是JavaScript兵器库中的一把瑞士军刀。从长远来看，细心学习、充分实践，将为你带来丰厚的长期回报。

3.8　数组

数组是值的有序集合。可以使用名称和索引来引用数组元素。在JavaScript中，有三种创建数组的方法：

```
var arr = new Array(1,2,3);
var arr = Array(1,2,3);
var arr = [1,2,3];
```

如果指定了值，数组就使用这些值作为元素来初始化。数组的length属性等于参数的个数。中括号语法称为数组字面量。由于写法简短，因而成为初始化数组的首选方法。

如果你想使用单个值为数字的元素初始化数组，就必须使用数组字面量语法。如果将单个数字值传给Array()构造函数或函数，JavaScript会将这个数字作为数组的长度，而非单个元素：

```
var arr = [10];
var arr = Array(10); // 创建一个没有实际元素的数组，将arr.length设为10
// 上面的代码等价于
var arr = [];
arr.length = 10;
```

JavaScript没有专门的数组类型。不过，你可以在程序中利用预定义的Array对象及其方法来使用数组。Array对象的方法能够以各种方式处理数组，例如合并、反转以及排序。另外，它有一个属性可以用来确定数组的长度，还有其他一些属性可以配合正则表达式使用。

可以通过给元素赋值来填充数组：

```
var days = [];
days[0] = "Sunday";
days[1] = "Monday";
```

也可以在创建数组的时候填充：

```
var arr_generic = new Array("A String", myCustomValue, 3.14);
var fruits = ["Mango", "Apple", "Orange"]
```

在大多数语言中，要求数组所有元素的类型必须相同。JavaScript允许数组中包含各种类型的值：

```
var arr = [
  'string', 42.0, true, false, null, undefined,
  ['sub', 'array'], {object: true}, NaN
];
```

可以使用元素的索引来引用数组元素。举例来说，假设你定义了以下数组：

```
var days = ["Sunday", "Monday", "Tuesday"]
```

可以使用colors[0]来引用第一个数组元素，使用colors[1]来引用第二个数组元素。元素索引从0开始。

JavaScript在内部是将数组元素作为标准的对象属性来存储的，数组索引就是属性名。属性length就不一样了，该属性的返回值总是比最后一个元素的索引多1。我们之前讲过，JavaScript数组索引是基于0的：索引从0开始，而不是从1，这意味着length属性值要比存储在数组中最高的索引值还要大1：

```
var colors = [];
colors[30] = ['Green'];
console.log(colors.length); // 31
```

你也可以给length属性赋值。如果赋予的值小于数组元素个数，数组会被截断；赋值0的话，会清空整个数组：

```
var colors = ['Red', 'Blue', 'Yellow'];
console.log(colors.length); // 3
colors.length = 2;
console.log(colors); // ["Red","Blue"] - Yellow已经被删除了
colors.length = 0;
console.log(colors); // [] colors数组为空
colors.length = 3;
console.log(colors); // [undefined, undefined, undefined]
```

如果查询的数组索引不存在，会返回undefined。

一个常见操作是迭代整个数组，处理每个元素。最简单的做法如下：

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

forEach()方法提供了另一种迭代数组的方法：

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
```

传递给forEach()的函数会针对数组中的每一个元素执行一次，在执行过程中，当前的数组元素会作为参数传入给该函数。forEach()循环不会迭代未分配值的元素。

Array对象有很多有用的方法，方法可以用来处理数组中的数据。

concat()方法能够合并两个数组，并返回合并后的新数组：

```
var myArray = new Array("33", "44", "55");
myArray = myArray.concat("3", "2", "1");
console.log(myArray);
// ["33", "44", "55", "3", "2", "1"]
```

join()方法能够将数组的所有元素合并成一个字符串。这个方法可用于处理列表。默认的分隔符是逗号（，）：

```
var myArray = new Array('Red','Blue','Yellow');
var list = myArray.join(" ~ ");
console.log(list);
//"Red ~ Blue ~ Yellow"
```


pop()方法能够删除数组最后一个元素并将其返回。它类似于栈的pop()方法：

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop();
// myArray = ["1", "2"], last = "3"
```

push()方法能够在数组尾部添加一个或多个元素，并返回最终的数组长度：

```
var myArray = new Array("1", "2");
myArray.push("3");
// myArray = ["1", "2", "3"]
```

shift()方法能够删除数组第一个元素并将其返回：

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift();
// myArray = ["2", "3"], first = "1"
```

unshift()方法能够在数组的头部添加一个或多个元素，并返回数组新的长度：

```
var myArray = new Array ("1", "2", "3");
myArray.unshift("4", "5");
// myArray = ["4", "5", "1", "2", "3"]
```

reverse()方法能够反转或者说颠倒数组元素——第一个数组元素变成最后一个，最后一个变成第一个：

```
var myArray = new Array ("1", "2", "3");
myArray.reverse();
// 颠倒数组，因此myArray = [ "3", "2", "1" ]
```

sort()方法能够对数组元素进行排序：

```
var myArray = new Array("A", "C", "B");
myArray.sort();
// 对数组排序，因此myArray = [ "A","B","c" ]
```

sort()方法使用一个可选的回调函数来定义如何比较元素。该函数会比较两个元素并返回三个值中的一个。我们来学习下面的函数。

- indexOf(searchElement[, fromIndex])：在数组中搜索searchElement，并返回第一次匹配的索引。

```
var a = ['a', 'b', 'a', 'b', 'a','c','a'];
console.log(a.indexOf('b')); // 1
// 现在再从上一次匹配位置开始向后搜索
console.log(a.indexOf('b', 2)); // 3
console.log(a.indexOf('1')); // -1, 'q' is not found
```

- lastIndexOf(searchElement[, fromIndex])：和indexOf()方法类似，只不过是从后向前搜索。

```
var a = ['a', 'b', 'c', 'd', 'a', 'b'];
console.log(a.lastIndexOf('b')); // 5
// 现在再从上一次匹配位置开始向前搜索
console.log(a.lastIndexOf('b', 4)); // 1
console.log(a.lastIndexOf('z')); // -1
```

现在我们已经深入学习了JavaScript的数组，接下来介绍一个神奇的库：Underscore.js（http://underscorejs.org/）。Underscore.js提供了一批极其有用的函数式编程辅助程序，能够让你的代码更加清晰和函数化。

假设你已经熟悉了Node.js，接下来通过npm来安装Underscore.js：

```
npm install underscore
```

将Underscore安装成Node模块之后，我们会把所有的例子放到一个.js文件中，在Node.js上运行该文件进行测试。也可以利用Bower来安装Underscore。

就像jQuery的$模块，Underscore也定义了一个_模块，你可以通过引用该模块来调用所有的函数。

把下面的代码输入到一个文本文件中，并将其命名为test_.js：

```
var _ = require('underscore');
function print(n){
  console.log(n);
}
_.each([1, 2, 3], print);
// 打印出1 2 3
```

不使用underscore库的each()，上面的代码可以写成这样：

```
var myArray = [1,2,3];
var arrayLength = myArray.length;
for (var i = 0; i < arrayLength; i++) {
  console.log(myArray[i]);
}
```

在这里所看到的就是一种强大的函数式构建方式，它使得代码更加优雅和简洁。传统的实现

方法显然太啰嗦了，很多语言（如Java）都因此受到诟病。这些语言如今也在逐步地接受函数式范式。作为一名JavaScript程序员，重要的是要尽可能地将这些理解融入自己的代码中。

前面例子中的each()函数对一系列元素进行迭代，将每个元素依次交给迭代函数。每次调用迭代函数的时候都会传入三个参数（元素、索引以及列表）。在上例中，each()函数对数组[1,2,3]进行迭代，对数组中的每个元素调用print函数，并将数组元素作为参数传入。这是对传统数组遍历方式的一种替代。

range()函数能够创建一个整数列表。如果忽略起始值的话，则默认为0，步长默认为1。如果你想生成负数区间，可以将步长设为负数：


```
var _ = require('underscore');
console.log(_.range(10));
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
console.log(_.range(1, 11));
//[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
console.log(_.range(0, 30, 5));
//[ 0, 5, 10, 15, 20, 25 ]
console.log(_.range(0, -10, -1));
//[ 0, -1, -2, -3, -4, -5, -6, -7, -8, -9 ]
console.log(_.range(0));
//[]
```

range()默认使用整数来填充数组，不过要点小手段的话，也是可以用其他数据类型的：

```
console.log(_.range(3).map(function () { return 'a' }) );
[ 'a', 'a', 'a' ]
```

用这种方法创建并初始化数组，既快速又方便。放在以前，这通常都是用传统的循环来实现的。

map()利用转换函数（transformation function）来映射列表中的每一个值，从而生成一个新的数组。考虑下面的例子：

```
var _ = require('underscore');
console.log(_.map([1, 2, 3], function(num){ return num * 3; }));
//[3,6,9]
```

reduce()函数能够将一系列值归约成单个值。初始状态由迭代函数传入，后续的每一步也由迭代函数返回。下面的例子给出了reduce()的用法：

```
var _ = require('underscore');
var sum = _.reduce([1, 2, 3], function(memo, num) {
  console.log(memo,num);return memo + num; }, 0);
console.log(sum);
```

在本例子中，console.log(memo, num)只是为了清晰地展示概念而已。代码输出如下：

```
0 1
1 2
3 3
6
```

最终的结果是1+2+3之和，也就是6。如你所见，有两个值被传入了迭代函数。在第一次迭代中，调用迭代函数时使用了两个值(0,1)——在调用reduce()函数时，memo的值是默认的，1就是列表中第一个元素的值。在函数中，我们对memo和num求和，然后返回中间变量sum，该变量将作为memo参数，由函数iterate()使用——memo中最后包含的就是累计的sum。这个概念对于理解如何用中间状态来计算出最终结果非常重要。

filter()函数会对整个列表进行迭代，并返回一个由所有通过条件测试的元素组成的数组。来看下面的例子：

```
var _ = require('underscore');
var evens = _.filter([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });
console.log(evens);
```

filter()函数的迭代函数返回一个真值。最终的evens数组中包含了满足测试条件的所有元素。

和filter()函数功能相反的是reject()函数。从名字中就可以看出，该函数会对列表进行迭代，并忽略满足测试条件的所有元素：

```
var _ = require('underscore');
var odds = _.reject([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0;});
console.log(odds);
//[ 1, 3, 5 ]
```

我们使用的代码和前一个例子中的一样，只不过是把filter()换成了reject()，而结果正好相反。

contains()是个挺有用的小函数，如果某个值存在于列表中，它会返回true，否则就返回false：

```
var _ = require('underscore');
console.log(_.contains([1, 2, 3], 3));
//true
```

invoke()是一个非常有用的函数，我已经渐渐爱上它了。该函数会针对列表中的每个元素调用一个特定的函数。打从知道invoke()起，我都不知道自己用过多少次了。来看下面的例子：

```
var _ = require('underscore');
console.log(_.invoke([[5, 1, 7], [3, 2, 1]], 'sort'));
//[ [ 1, 5, 7 ], [ 1, 2, 3 ] ]
```

在这个例子中，对数组中的每个元素都调用了Array对象的sort()方法。注意，下面的写法是不行的：

```
var _ = require('underscore');
console.log(_.invoke(["new","old","cat"], 'sort'));
//[ undefined, undefined, undefined ]
```

这是因为sort方法并不是String对象的一部分。这样写就没问题：

```
var _ = require('underscore');
console.log(_.invoke(["new","old","cat"], 'toUpperCase'));
//[ 'NEW', 'OLD', 'CAT' ]
```

这是因为toUpperCase()是String对象的方法，列表中所有的元素也都是String类型。

uniq()函数会删除原始数组中所有重复的元素，并返回该数组：

```
var _ = require('underscore');
var uniqArray = _.uniq([1,1,2,2,3]);
console.log(uniqArray);
//[1,2,3]
```


partition()函数能够将数组一分为二：其中一个数组中的元素都能够满足谓词（predicate），不能满足的都在另一个数组中：

```
var _ = require('underscore');
function isOdd(n){
  return n%2==0;
}
console.log(_.partition([0, 1, 2, 3, 4, 5], isOdd));
//[ [ 0, 2, 4 ], [ 1, 3, 5 ] ]
```

compact()返回一个数组的副本，其中不包括所有的假值（false、null、0、""、undefined和NaN）：

```
console.log(_.compact([0, 1, false, 2, '', 3]));
```

这段代码会删除所有的假值，返回一个包含了元素[1,2,3]的新数组——该方法有助于从列表中消除会造成运行时异常的值。

without()函数会返回一个删除了所有指定值的数组的副本：

```
var _ = require('underscore');
console.log(_.without([1,2,3,4,5,6,7,8,9,0,1,2,0,0,1,1],0,1,2));
//[ 3, 4, 5, 6, 7, 8, 9 ]
```

3.9 map

ECMAScript 6引入了map。map是一个简单的键值映射，可以依据元素的插入顺序进行迭代。下面的代码片段展示了Map类型的一些方法及其用法：

```
var founders = new Map();
founders.set("facebook", "mark");
founders.set("google", "larry");
founders.size; // 2
founders.get("twitter"); // undefined
```

```
founders.has("yahoo"); // false

for (var [key, value] of founders) {
  console.log(key + " founded by " + value);
}
// "facebook founded by mark"
// "google founded by larry"
```

3.10　set

ECMAScript 6引入了set。set是值的集合，可以依据其插入顺序进行迭代。set的一个重要特点就是其中的值只能出现一次。

下面的代码片段展示了set的一些基本操作：

```
var mySet = new Set();
mySet.add(1);
mySet.add("Howdy");
mySet.add("foo");

mySet.has(1); // true
mySet.delete("foo");
mySet.size; // 2

for (let item of mySet) console.log(item);
// 1
// "Howdy"
```

我们简要地讨论过，JavaScript的数组并非传统意义上的数组。在JavaScript中，数组作为对象，具备以下特点：

- length属性
- 从Array.prototype处继承函数（将在下一章中讨论）
- 对数字类型的键会做特殊处理

当我们将数组索引写成数字时，索引会被转换成字符串——arr[0]在内部会变成arr["0"]。因此，在使用JavaScript数组时，有几件事要注意。

- 通过索引访问数组元素并不像在C语言中那样是一种常数时间操作。因为数组实际上就是键值的映射，所以具体的访问要依赖于映射的分布（layout）及其他因素（冲突等）。
- JavaScript数组是稀疏的（大部分元素都有默认值），这意味着数组中可以存在间隙。要理解这一点，请看下面的代码片段：

  ```
  var testArr=new Array(3);
  console.log(testArr);
  ```

你会看到输出结果是[undefined, undefined, undefined]——undefined是数组元素的默认值。

考虑这个例子：

```
var testArr=[];
testArr[3] = 10;
testArr[10] = 3;
console.log(testArr);
// [undefined, undefined, undefined, 10, undefined, undefined, undefined, undefined,
undefined, undefined, 3]
```

可以看到数组中存在着一些间隙。只有两个元素有具体的内容，其他的都是包含默认值的间隙。了解这一点是有好处的。使用for...in循环迭代数组会出现意想不到的结果。看下面的例子：

```
var a = [];
a[5] = 5;
for (var i=0; i<a.length; i++) {
  console.log(a[i]);
}
// 对索引0到5的元素进行迭代
// [undefined,undefined,undefined,undefined,undefined,5]

for (var x in a) {
  console.log(x);
}
// 只显示索引为5的元素，忽略索引0-4
```

3.11 编码风格

和之前一样，我们来花点时间讨论一下创建数组时的写法。

- 使用字面量语法创建数组：

```
// 不建议
const items = new Array();
// 建议
const items = [];
```

- 使用数组的push()方法添加数组元素，而不是选择直接赋值：

```
const stack = [];
// 不建议
stack[stack.length] = 'pushme';
// 建议
stack.push('pushme');
```

3.12 小结

随着JavaScript语言的成熟，其工具链也变得愈发稳健和高效，很少看到编程老手不用Underscore.js这种库。随着学习的深入，我们会继续探索更多功能强大的库，在它们的帮助下，你的代码会变得更紧凑、可读性更好、性能更高。我们学习了正则表达式——JavaScript中的头等对象。一旦理解了`RegExp`，就会发现多使用它，能够让你的代码更简洁。在下一章中，我们将学习JavaScript中Object的写法及其原型继承的来龙去脉，并将后者作为一种审视面向对象编程的全新方式。

第 4 章

面向对象的JavaScript

JavaScript最基本的数据类型是Object。JavaScript对象可视为可修改（mutable）的键值集合。在JavaScript中，数组、函数和正则表达式都是对象，而数字、字符串和布尔值则为类似于对象的语言构件，它们不可修改，但是拥有方法[①]。本章将学习以下主题：

- 理解对象
- 实例属性与原型属性
- 继承
- 接收器与设置器

4.1 理解对象

在开始学习JavaScript如何处理对象之前，有必要先讨论一下面向对象范式。与大多数编程范式一样，**面向对象编程**（OOP）也是源于复杂性管理的需要。其主要理念是将整个系统划分成规模更小的部分，各部分之间彼此独立。如果这些较小的部分能够尽可能多地隐藏实现细节，它们就会变得易于使用。有一个关于汽车的经典比喻有助于你理解OOP这一极为重要的概念。

当驾驶一辆汽车时，你操作的是接口——方向盘、离合器、刹车、油门。你只能通过这些接口开操作这辆车；有了它们，我们才能驾驶这辆汽车。接口实际上隐藏了所有真正驱动汽车的复杂系统，例如引擎的内部功能、电子系统等。作为一名司机，这些复杂性不用你来操心。OOP的主要驱动力类似于此。对象隐藏了特定功能复杂的实现细节，只向外部暴露有限的接口，其他系统可以使用这些接口而无需关注内部所隐藏的复杂性。另外，对象通常会隐藏内部状态，避免其他对象直接修改。这就是OOP的要点。

在大型系统中，大量的对象调用其他对象的接口。如果你允许它们修改对方的内部状态，那可就麻烦了。OOP的操作是这样的：对象的状态对于外部而言是隐藏的，只能够通过受控的接口

① 实际上，具有方法的应该是这些原始类型所对应的包裹对象（wrapper object）。参见O'Reilly出版社的*JavaScript: The Definitive Guide*第6版中的3.6节“Wrapper Objects”。——译者注

操作来改变。

OOP是一个重要的概念，较传统的结构化编程而言的确是一种进步。然而，很多人觉得OOP做得过头了。大多数OOP系统都定义了复杂且毫无必要的类和类型的继承。另一个重大的缺陷是推崇状态的隐藏，OOP认为对象的状态基本上无足轻重。尽管OOP相当流行，但在很多领域，其缺点也是一目了然的。当然，OOP的有些理念还是挺不错的，尤其是隐藏复杂性、只向外部暴露接口。JavaScript汲取了一些有益的概念，并围绕其建立了自己的对象模型。幸运的是，这个设计决定使得JavaScript对象神通广大。“四人组”（Gang of Four）[①]在其重要著作《设计模式：可复用的面向对象软件基础》中，提出了更好的面向对象设计的两条基本原则：

- 针对接口编程，而非实现（Program to an interface and not to an implementation）
- 对象的组合优于类的继承（Object composition over class inheritance）

这两个概念的确与典型的OOP运作方式背道而驰。继承的典型操作方式就是基于继承，将父类暴露给所有的子类。在典型的继承中，子类和父类是紧密耦合在一起的。有一些机制能够在一定程度上解决这个问题。如果你是在Java中使用这种典型的继承方式，通常建议**针对接口编程，而非实现**。在Java中，可以使用接口来编写解耦代码：

```
// 对List接口编程，而非针对ArrayList实现
List theList = new ArrayList();
```

我们不针对具体的实现编程，而是执行以下操作：

```
ArrayList theList = new ArrayList();
```

如何对接口编程？在对`List`接口编程的时候，你只能够调用`List`接口可用的方法，`ArrayList`的特定方法是没法调用的。针对接口编程能够让你自由地改变自己的代码，并使用`List`接口的任何其他特定子接口。例如，我可以修改我的实现，使用`LinkedList`而非`ArrayList`。你可以修改变量来使用`LinkedList`：

```
List theList = new LinkedList();
```

这种方法的优美之处在于，就算你在程序中的100个地方使用了`List`，也不用担心得去修改所有位置上的实现。当你践行了“针对接口编程，而非实现”的原则时，就能够写出松耦合的代码了。在使用典型的继承方式时，这是一个重要的原则。

在典型继承中还有一处限制：只能在父类的限制下增强子类。你没法从根本上改变从祖先那里继承来的东西，这就无法实现重用了。典型的继承还有如下两个问题。

- 继承导致了紧耦合。子类明悉他们的祖先，这就使得子类与其父类紧紧耦合在一起。

① “四人组”指的是《设计模式》一书的四名作者：Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides。——译者注

- 当从父类中生成子类时，你没有选择继承什么、不继承什么的权利。Erlang之父Joe Armstrong对此有一段论述，非常好地解释了这种情形："面向对象语言的问题在于其自身就已经包含并携带了全部的隐含环境。你要的是一根香蕉，得到的却是拿着香蕉的大猩猩以及整片丛林。"

4.1.1　JavaScript 对象的行为

有了这些背景知识，现在让我们来探究JavaScript对象的行为方式。从广义上来说，一个对象可以包含属性，属性被定义为键值对。属性的键（名称）可以是一个字符串，值可以是任何有效的JavaScript值。可以使用对象的字面形式来创建对象。下面的代码片段展示了如何创建对象：

```
var nothing = {};
var author = {
  "firstname": "Douglas",
  "lastname": "Crockford"
}
```


属性名可以是任意的字符串或空串。如果属性名是合法的JavaScript名称的话，可以省略其周围的引号。因此，对于`first-name`来说，引号是必需的；对于`firstname`来说，引号是可选的。逗号用于分隔键值对。对象也可以嵌套：

```
var author = {
  firstname : "Douglas",
  lastname : "Crockford",
  book : {
    title:"JavaScript- The Good Parts",
    pages:"172"
  }
};
```

访问对象属性有两种写法：一种写法类似于访问数组元素，另一种使用点号。在第一种写法中，需要将一个字符串表达式放入`[]`中来获取对象属性值。如果表达式是一个有效的JavaScript名称的话，也可以使用点号写法，这种写法是检索属性值的首选：

```
console.log(author['firstname']); //Douglas
console.log(author.lastname);     //Crockford
console.log(author.book.title);   // JavaScript- The Good Parts
```

如果试图检索一个不存在的属性，会导致`undefined`错误：

```
console.log(author.age);
```

在这种情况下，一个技巧是使用`||`操作符来填充默认值：

```
console.log(author.age || "No Age Found");
```

你可以给属性赋以新值来更新对象：

```
author.book.pages = 190;
console.log(author.book.pages); //190
```

仔细观察，会发现对象的字面形式语法和JSON格式很像。

方法是一个具有函数值的对象的属性：

```
var meetingRoom = {};
meetingRoom.book = function(roomId){
  console.log("booked meeting room -"+roomId);
}
meetingRoom.book("VL");
```

4.1.2　原型

除了我们添加给对象的属性之外，几乎所有的对象都有一个叫作prototype的默认属性。如果对象没有所请求的属性，JavaScript会到原型中去查找。`Object.getPrototypeOf()`方法会返回对象的原型。

很多程序员认为原型与对象的继承紧密相关——这的确是定义对象类型的一种方法——但从根本上而言，与原型密切相关的是函数。

原型是定义能够被对象实例所使用的属性和函数的一种方式。原型的属性最终会成为对象实例的属性。原型可视为对象创建的模板，它类似于面向对象语言中的类。JavaScript中的原型可用来编写典型的面向对象代码以及模拟典型的继承形式。让我们回顾一下先前的例子：

```
var author = {};
author.firstname = 'Douglas';
author.lastname = 'Crockford';
```

在上面的代码片段中，我们创建了一个新对象并分别赋予其属性。你很快就会意识到，这并不是一个创建对象的标准方法。如果你对OOP有所了解，就会发现这里既没有封装，也没有通常的类结构。JavaScript对此有相应的解决方法。可以使用`new`操作符，通过构造函数来实例化一个对象。不过，在JavaScript中并没有类的概念，另外要注意的是`new`操作符是应用于构造函数的，这一点很重要。为了清楚地理解这一点，让我们来看看下面的例子：

```
// 一个什么都不返回，也什么都不创建的函数
function Player() {}
  // 向该函数的prototype属性中添加一个函数
  Player.prototype.usesBat = function() {
    return true;
  }

  // 以函数的形式调用player()，以证明什么都不会发生
  var crazyBob = Player();
  if(crazyBob === undefined){
    console.log("CrazyBob is not defined");
```

```
  }

  // 显示使用new，以构造函数的形式调用player()
  // 1. 创建实例
  // 2. 从函数的原型中得到方法usesBat()
  var swingJay = new Player();
  if(swingJay && swingJay.usesBat && swingJay.usesBat()){
    console.log("SwingJay exists and can use bat");
}
```

在上例中，`player()`函数什么都不做。我们采用了两种不同的方法来调用该函数，一种是按照普通函数来调用，另一种是作为构造函数来调用——注意其中使用了`new`操作符。函数定义好之后，我们向其添加了一个`userBat()`方法。当`player()`以普通函数调用时，对象并没有被实例化，`undefined`被赋给了`crazyBob`。但当使用`new`操作符调用该函数的时候，就得到了一个完整的实例化对象`swingJay`。

4.2 实例属性与原型属性

实例属性是作为对象实例自身一部分的那些属性，如下例所示：

```
function Player() {
  this.isAvailable = function() {
    return "Instance method says - he is hired";
  };
}
Player.prototype.isAvailable = function() {
  return "Prototype method says - he is Not hired";
};
var crazyBob = new Player();
console.log(crazyBob.isAvailable());
```

运行这个例子，你会看到输出了字符串`Instance method says - he is hired`。在函数`Player()`中定义的函数`isAvailable()`作为`Player`实例的方法被调用。这意味着除了将属性附着在原型上之外，也可以使用关键字`this`在构造函数中初始化属性。如果相同的函数同时被定义为实例属性和原型属性，则实例属性的优先级更高。控制初始化优先级的规则如下：

- 原型中的属性被绑定到对象实例中
- 构造函数中的属性被绑定到对象实例中

本例中为我们展示了关键字`this`的用法。这个关键字的用法很容易把人搞糊涂，因为它在JavaScript中的行为并不一致。在Java等其他面向对象语言中，关键字`this`指向的是类的当前实例。在JavaScript中，`this`的值是由函数的调用上下文（invocation context）以及调用位置所决定的。让我们来看看如何理解这种行为。

- 当`this`用于全局上下文中：如果是在全局上下文中使用`this`，它就会被绑定在全局上下

文。比如在浏览器中，全局上下文通常是window。对于函数来说也是如此。如果是在定义在全局上下文的函数中使用this，它依然被绑定在全局上下文，因为该函数是全局上下文的一部分：

```
function globalAlias(){
  return this;
}
console.log(globalAlias()); //[object Window]
```

- 当this用于对象方法中：在这种情况下，this被赋值或绑定到包含对象（enclosing object）上。注意，如果对象存在嵌套的话，包含对象就是那个直接的父对象（immediate parent）：

```
var f = {
  name: "f",
  func: function () {
    return this;
  }
};
console.log(f.func());
//prints -
//[object Object] {
//  func: function () {
//    return this;
//  },
//  name: "f"
//}
```

- 如果不存在上下文：当一个函数不跟随任何对象调用的时候，就不会有上下文。默认情况下，它会被绑定到全局上下文。如果在这种函数中使用this，它也会被绑定到全局上下文。
- 当this用于构造函数中：我们之前已经看到过，当函数使用关键字new来调用时，它就会被作为构造函数使用。在构造函数中，this指向被构造的对象。在下面的例子中，f()是构造函数（因为是通过关键字new调用的），因此this指向的就是被创建出的新对象。当我们使用this.member = "f"时，新的member属性就被添加到新创建的对象中，在这个例子中，新对象就是o：

```
var member = "global";
function f()
{
  this.member = "f";
}
var o= new f();
console.log(o.member); // f
```

我们知道如果实例和原型中都定义了相同的属性，会优先使用实例属性。很容易想象当创建出一个新对象时，将构造函数原型[①]的属性复制过来。不过这种假设是不对的。实际所发生的是

① 这里指的是构造函数的prototype属性（property），而非prototype特性（attribute）。——译者注

将原型附着在对象上，在该对象属性被引用的时候去原型中查找。基本上，当引用到对象属性时，会发生以下操作之一。

- 检查对象是否拥有该属性。如果有的话，返回该属性的值。
- 检查与对象相关联的原型。如果在其中找到了该属性，将其返回；否则，返回undefined错误。

理解这一点非常重要，因为下面的代码在JavaScript中运行起来毫无问题：

```
function Player() {
  isAvailable=false;
}
var crazyBob = new Player();
Player.prototype.isAvailable = function() {
  return isAvailable;
};
console.log(crazyBob.isAvailable()); //false
```


这段代码与之前的例子略有不同。我们首先创建了一个对象，然后将一个函数附加到其原型。当最后在对象上调用isAvailable()方法时，如果在该对象（本例中是crazyBob）中没有找到这个方法，JavaScript会去对象原型中搜索。这就像是**代码实时加载**（hot code loading）——如果运用得当，这种能力在扩展基本的对象框架方面具有不可思议的威力，就算是在对象创建之后也是如此。

如果你熟悉OOP，肯定想知道是否能够控制对象成员的可见性及其访问方式。我们之前讲过，JavaScript中没有类。在像Java这样的编程语言中，你可以使用诸如private和public这种访问修饰器（access modifier）来控制类成员的可见性。而在JavaScript中，我们可以利用函数作用域来实现类似的效果。

- 可以在函数中使用关键字var来声明私有变量，它们可以由私有函数或特权方法访问。
- 可以在对象的构造函数中声明私有函数，并由特权方法调用。
- 特权方法可以使用this.method = function() {}来声明。
- 公共方法可以使用Class.prototype.method = function() {}来声明。
- 公共属性可以使用this.property来声明，能够在对象外部访问。

下面的例子展示了一些实现方法：

```
function Player(name,sport,age,country){

  this.constructor.noOfPlayers++;

  // 私有属性和函数
  // 只能由特权对象浏览、编辑或调用
  var retirementAge = 40;
  var available=true;
  var playerAge = age?age:18;
```

```
  function isAvailable(){ return available && (playerAge<retirementAge); }
  var playerName=name ? name :"Unknown";
  var playerSport = sport ? sport : "Unknown";

  // 特权方法
  // 可以从外部调用，也可以由成员访问
  // 可以替换成对应的公共方法
  this.book=function(){
    if (!isAvailable()){
      this.available=false;
    } else {
      console.log("Player is unavailable");
    }
  };
  this.getSport=function(){ return playerSport; };
  // 公共属性，可以在任何地方对其作出修改
  this.batPreference="Lefty";
  this.hasCelebGirlfriend=false;
  this.endorses="Super Brand";
  }

// 公共方法——任何人都可以读取或写入
// 只能访问公共属性和原型属性
Player.prototype.switchHands = function(){ this.batPreference="righty"; };
Player.prototype.dateCeleb = function(){ this.hasCelebGirlfriend=true; } ;
Player.prototype.fixEyes = function(){ this.wearGlasses=false; };

// 原型属性——任何人都可以读取或写入（或覆盖）
Player.prototype.wearsGlasses=true;

// 静态属性——任何人都可以读取或写入
Player.noOfPlayers = 0;

(function PlayerTest(){
  // 创建Player对象的新实例
  var cricketer=new Player("Vivian","Cricket",23,"England");
  var golfer =new Player("Pete","Golf",32,"USA");
  console.log("So far there are " + Player.noOfPlayers + " in the guild");
  // 两个函数共享公共的Player.prototype.wearsGlasses变量
  cricketer.fixEyes();
  golfer.fixEyes();

  cricketer.endorses="Other Brand";// 可以更新公有变量

  // 通过Player的原型来改变其公共方法
  Player.prototype.fixEyes=function(){
   this.wearGlasses=true;
  };
  // 只改变了Cricketer的函数
  cricketer.switchHands=function(){
    this.batPreference="undecided";
  };

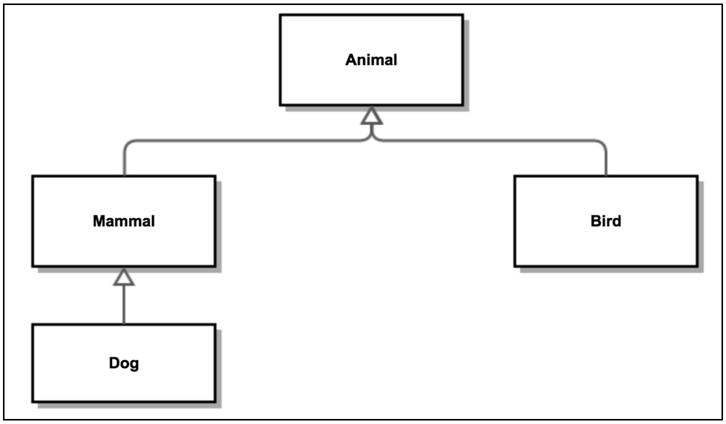
})();
```

我们需要理解这个例子中的几个重要概念。

- retirementAge是一个私有变量，没有特权方法能够获取或设置该变量的值。
- country变量是作为构造函数参数而创建的私有变量。构造函数参数可作为对象的私有变量。
- 当调用cricketer.switchHands()时，它只能应用于cricketer，不能用于golfer，尽管它是Player的原型函数。
- 私有函数和特权方法会在创建出新的对象时实例化。本例中，每个新的Player实例都有自己的isAvailable()和book()副本。而公共方法只有一份，在所有实例之间共享，这样对性能会有一点好处。如果你确实不需要什么私有内容，可以考虑为其设置公共属性。

4.3 继承

继承是OOP中一个重要的概念。经常会看到一堆对象都实现了相同的方法。除了少数几个方法之外，几乎会看到一模一样的对象定义，这种情况也很常见。继承在提高代码重用率方面效果显著。下面来看一个继承关系的典型例子：



可以从一般性的Animal类中看到，我们基于一些特征衍生出了如Mammal和Bird这种更具体的类。Mammal和Bird类与Animal类有着共同之处，但除此之外，它们也有自己独特的行为和属性。最终，我们衍生出了更为具体的哺乳动物：Dog类。Dog类与Animal类和Mammal类有共同的属性和行为，但自身又有着属于狗的属性和行为。这个过程可以继续进行，从而获得更复杂的继承关系。

传统上，继承用于建立或描述一种IS-A（是一个）的关系。例如，狗IS-A哺乳动物。这就是所谓的**类继承**（classical inheritance）。你可以在如C++或Java这种面向对象语言中看到这样的关系。JavaScript采用了一种完全不同的机制来处理继承。作为一种无类的语言，JavaScript利用原型来

实现继承。和类继承相比，原型继承在性质上大不相同，需要全面的考察理解。这两种继承方法存在很大差异，必须仔细研究。

在类继承中，实例从类模板（class blueprint）中继承并创建子类关系。你无法在类定义上调用实例方法，只能先创建出实例，然后在该实例上调用方法。在原型继承中，实例从其他实例中继承。

就继承而言，JavaScript只使用对象。我们之前讨论过，每个对象都有一个链接，指向另一个叫作原型的对象。这个原型对象也有自己的原型，以此类推，直到出现一个原型为`null`的对象。当然，`null`没有原型，它作为原型链中的最后一环。

为了更好地理解原型链，来看下面的例子：

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
function Child() {}
Child.prototype = {cry: Person.prototype.cry};
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //false
console.log(aChild instanceof Object); //true
```

这里，我们定义了`Person`和`Child`——child（孩子）IS-A person（人），另外还将`Person`的`cry`属性复制到了`Child`的`cry`属性。当我们使用`instanceOf`观察两者之间的关系时，很快就会发现单凭复制行为无法真正使得`Child`变成`Person`的实例，`aChild instanceof Person`结果为假。这只不过是复制，或者说是伪装（masquerading），并不是继承。即便是将`Person`的所有属性都复制给`Child`，这也并非继承。在此展示这种不良做法仅仅是出于讲解的目的。我们希望有一个原型链——一种IS-A的关系，一种真正的继承，可以让我们说child（孩子）IS-A person（人）。我们要创建一个链：child（孩子）IS-A person（人）IS-A mammal（哺乳动物）IS-A animal（动物）IS-A object（对象）。在JavaScript中，可以通过将对象的实例作为原型来实现：

```
SubClass.prototype = new SuperClass();
Child.prototype = new Person();
```

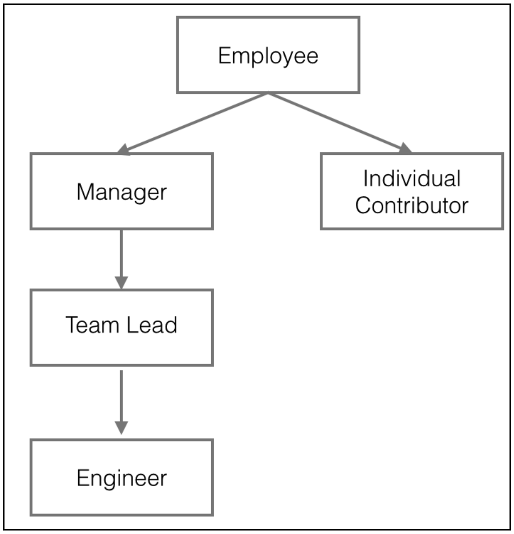
修改一下先前的例子：

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
function Child() {}
Child.prototype = new Person();
var aChild = new Child();
console.log(aChild instanceof Child); //true
console.log(aChild instanceof Person); //true
```

```
console.log(aChild instanceof Object); //true
```

修改过的那行代码使用Person的实例作为Child的原型。和之前的方法相比，这是一处重要的差异。我们以此说明child（孩子）IS-A person（人）。

我们讲过JavaScript沿原型链向上查找属性，直到遇见Object.prototype。下面我们将详细讨论原型链的概念，尝试设计出下面的雇员层次结构：



这是一个典型的继承模式。经理和雇员是IS-A(n)关系，Manager包含从Employee处继承得来的公共属性。前者会有一些直属下级。Individual Contributor同样基于Employee，但是没有直属下级。从Manager衍生出的Team Lead包含一些Manager所不具备的功能。我们所做的就是让每一个子类从父类中获得属性（Manager与Team Lead是父子关系）。

来看看如何在JavaScript中创建这种层次结构。先来定义Employee类型：

```
function Employee() {
  this.name = '';
  this.dept = 'None';
  this.salary = 0.00;
}
```

这些定义没有什么特别之处。Employee对象包含了三个属性：name、salary和department。接下来定义Manager。这个定义展示了如何指定继承链中的下一个对象：

```
function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);
```

在JavaScript中，可以将一个原型实例作为构造函数的prototype属性的值。在定义构造函数之后，你可以随时这么做。在本例中，有两处做法之前并没有碰到过。首先，我们调用了Employee.call(this)。如果你有Java背景的话，这类似于在构造函数中调用super()方法。call()方法调用函数时使用指定的对象作为函数上下文（在这里，这个对象会作为this的值），换句话说，call()允许指定函数运行时关键字this指向哪个对象。就像Java中的super()，必须调用parentObject.call(this)来正确地初始化所创建的对象。

在另一个地方，我们并没有调用new()，而是使用了Object.create()。Object.create()在创建对象时可以指定原型。在执行new Parent()时，会调用对应的构造函数。在大多数情况下，我们希望Child.prototype是一个能够通过其原型链接到Parent.prototype上的对象。如果Parent的构造函数包含了其他针对于Parent类的逻辑，我们并不想在创建Child对象时执行它们，这会导致非常难以排查的错误。Object.create()可以在不用调用Parent构造函数的情况下，在父子之间创建和使用new操作符时一样的原型链。

要想拥有一个既没有副作用又准确无误的继承机制，必须确保执行以下操作。

- 将原型设置成父类的实例来初始化原型链（继承）。这一步只需要做一次（因为原型对象是共享的）。
- 调用父类的构造函数来初始化对象自身。在每次实例化的时候都要这么做（每次构造对象的时候可以传入不同的参数）。

理解了这两点之后，让我们来定义余下的对象：

```
function IndividualContributor() {
  Employee.call(this);
  this.active_projects = [];
}
IndividualContributor.prototype = Object.create(Employee.prototype);

function TeamLead() {
  Manager.call(this);
  this.dept = "Software";
  this.salary = 100000;
}
TeamLead.prototype = Object.create(Manager.prototype);

function Engineer() {
  TeamLead.call(this);
  this.dept = "JavaScript";
  this.desktop_id = "8822" ;
  this.salary = 80000;
}
Engineer.prototype = Object.create(TeamLead.prototype);
```

根据这样的层次结构，我们可以实例化这些对象：

```
var genericEmployee = new Employee();
console.log(genericEmployee);
```

上面的代码片段输出如下：

```
[object Object] {
  dept: "None",
  name: "",
  salary: 0
}
```

一般的Employee所属部门（department）被赋值为None（默认值），其余的属性也被赋予同样的默认值。

接下来要做的是实例化Manager。可以指定下列属性值：

```
var karen = new Manager();
karen.name = "Karen";
karen.reports = [1,2,3];
console.log(karen);
```


以上代码输出如下：

```
[object Object] {
  dept: "None",
  name: "Karen",
  reports: [1, 2, 3],
  salary: 0
}
```

对于TeamLead，其reports属性从基类获得（在本例中是Manager）：

```
var jason = new TeamLead();
jason.name = "Json";
console.log(jason);
```

以上代码输出如下：

```
[object Object] {
  dept: "Software",
  name: "Json",
  reports: [],
  salary: 100000
}
```

JavaScript处理new操作符时，会创建一个新对象，然后将该对象作为this的值传给构造函数TeamLead()。该构造函数会设置projects属性的值，并悄悄地将内部属性_prototype_设置成TeamLead.prototype的值。_prototype_属性决定了用于返回属性值的原型链。这个过程并不会在对象jason中设置从原型链继承得来的属性值。当需要读取某个属性的值时，JavaScript首先会检查是否能在该对象中找到这个属性的值。如果找到，就返回属性值。如果找不到，JavaScript就会利用_prototype_属性到原型链中去查找。照这样说的话，执行下面的语句会怎样呢？

```
Employee.prototype.name = "Undefined";
```

这并不会传播（propagate）到Employee的所有实例中。这是因为在创建Employee实例的时候，实例已经有了该同名属性（name）的本地值。当通过创建新的Employee对象来设置TeamLead的原型的时候，TeamLead.prototype也有了name属性的本地值。因此，在JavaScript查找jason对象（TeamLead的实例）的name属性时，会在TeamLead.prototype中找该属性的本地值，它不会到Employee.prototype中做进一步搜索。

如果你想在运行时修改属性值，并希望该对象所有的后代能够继承这个值，那就别把属性定义到对象的构造函数中。要做到这一点，需要把属性添加到构造函数的原型中[①]。让我们回顾一下之前的例子并稍作修改：

```
function Employee() {
  this.dept = 'None';
  this.salary = 0.00;
}
Employee.prototype.name = '';
function Manager() {
  this.reports = [];
}
Manager.prototype = new Employee();
var sandy = new Manager();
var karen = new Manager();

Employee.prototype.name = "Junk";

console.log(sandy.name);
console.log(karen.name);
```

你会发现sandy和karen的name属性都变成了Junk，这是因为name属性是在构造函数之外声明的。因此，当修改了Employee原型中name属性的值，会将其传播到所有的后代。在本例中，我们是在创建sandy和karen对象之后修改了Employee的原型。如果你是在sandy和karen对象创建之前修改的原型，值仍然会变成Junk。

所有的JavaScript原生对象——Object、Array、String、Number、RegExp以及Function——都具有可以扩展的prototype属性，这意味着我们可以扩展语言本身的功能。例如，下面的代码片段就扩展了String对象，为其添加了一个能够反转字符串的reverse()方法。在原生String对象中并没有这个方法，但是通过处理String对象的原型，就可以添加上这个方法：

```
String.prototype.reverse = function() {
  return Array.prototype.reverse.apply(this.split('')).join('');
};
var str = 'JavaScript';
console.log(str.reverse()); //"tpircSavaJ"
```

① 这里指的是构造函数的prototype属性所指向的原型对象，并非是构造函数的[[prototype]]。——译者注

尽管这是一项非常强大的技术，但切记不要滥用。可以参阅http://perfectionkills.com/extending-native-builtins/来了解原生内建对象扩展过程中的陷阱以及操作过程中需要留意的事项。

4.4　接收器与设置器

接收器（getter）是一种很方便的方法，可以用来获取特定属性的值；和名字一样，**设置器**（setter）是可以用来设置属性值的方法。你可能经常需要根据其他一些值来生成某个值。接收器和设置器通常会写成如下形式的函数：

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  setLastName: function(_lastname){
    this.lastname= _lastname;
  },
  setFirstName: function (_firstname){
    this.firstname= _firstname;
  },
  getFullName: function (){
    return this.firstname + ' '+ this.lastname;
  }
};
person.setLastName('Newton');
person.setFirstName('Issac');
console.log(person.getFullName());
```

如你所见，`setLastName()`、`setFirstName()`和`getFullName()`都是用来接收（get）和设置（set）属性的函数。`Fullname`是通过拼接`firstName`和`lastName`属性而生成的属性。这种用法很常见，ECMAScript 5如今为接收器和设置器提供了默认语法。

下面的例子展示了在ECMAScript 5中如何使用对象字面量语法来创建接收器和设置器：

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  get fullname() {
    return this.firstname +" "+this.lastname;
  },
  set fullname(_name){
    var words = _name.toString().split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
};
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

另一种声明接收器和设置器的方法是使用`Object.defineProperty()`：

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
};
Object.defineProperty(person, 'fullname', {
  get: function() {
    return this.firstname + ' ' + this.lastname;
  },
  set: function(name) {
    var words = name.split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
});
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

在这种方法中，就算对象已经创建，你也可以调用`Object.definePropery()`。

现在你已经了解了JavaScript面向对象方面的特点，接下来要学习一些由Underscore.js所提供的非常有用的工具方法。Underscore.js的安装以及用法在上一章中已经讲过了。这些方法可以让一些对象的日常操作变得轻而易举。

- `keys()`：该方法能够检索对象可枚举的自有属性名称。注意，它并不会向上遍历原型链：

```
var _ = require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
};
console.log(_.keys(testobj));
//[ 'name', 'age', 'profession' ]
```

- `allKeys()`：该方法能够检索对象自有的以及继承的属性名称。

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
}
Scientist.prototype.married = true;
aScientist = new Scientist();
console.log(_.keys(aScientist)); //[ 'name' ]
console.log(_.allKeys(aScientist));//[ 'name', 'married' ]
```

- `values()`：该方法能够检索对象自有属性的值。

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
}
```

```
Scientist.prototype.married = true;
aScientist = new Scientist();
console.log(_.values(aScientist)); //[ 'Albert' ]
```

- mapObject()：该方法能够转换对象中各个属性的值。

```
var _ = require('underscore');
function Scientist() {
  this.name = 'Albert';
  this.age = 90;
}
aScientist = new Scientist();
var lst = _.mapObject(aScientist, function(val,key){
  if(key==="age"){
    return val + 10;
  } else {
    return val;
  }
});
console.log(lst); //{ name: 'Albert', age: 100 }
```

- functions()：返回一个包含对象中所有方法名称的有序列表，也就是对象所有的函数名。
- pick()：返回一个只包含指定属性的对象副本。

```
var _ =require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
};
console.log(_.pick(testobj, 'name','age')); //{ name: 'Albert', age: 90}
console.log(_.pick(testobj, function(val,key,object){
  return _.isNumber(val);
})); //{ age: 90 }
```

- omit()：功能和pick()相反，它返回一个不包含指定属性的对象副本。

4.5 小结

利用面向对象，我们可以更大程度地控制代码及其结构，借此提高JavaScript应用程序的条理性以及质量。JavaScript的面向对象基于的是函数原型和原型继承，这两个概念为开发人员提供了异常丰富的灵感源泉。

在本章中，我们学习了对象创建和处理的基本方法，了解了构造函数创建对象的方法，还深入研究了原型链以及基于原型链的继承方法，这些基础知识可用于构建下一章将要讲到的JavaScript模式。

第 5 章

JavaScript模式

5

到目前为止，我们已经学习了编写JavaScript代码所需的基础知识。一旦开始使用这些基本构件来构造规模更大的系统，你很快就会意识到有些事情是有标准做法的。在开发大型系统的时候，你会碰到一些不断重复出现的问题；模式就是为这种已知且明晰的问题提供标准化解决方案的。可以将模式视为一种最佳实践、一种有价值的抽象或是一种解决常见问题的模板。编写可维护的代码并非易事。写出模块化、正确且可维护代码的关键在于能够理解那些重复出现的主题，利用通用的模板设计出优化的解决方案。关于设计模式最重要的文献就是《设计模式：可复用的面向对象软件基础》，作者是被称为“四人组”（GOF）的Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides。这本学术著作给出了各种模式的正式定义，解释了现今流行使用的大部分模式的实现细节。关键是要理解为什么设计模式如此重要。

- 模式为常见问题提供了行之有效的解决方案：模式提供了解决特定问题的优化模板。坚实深厚的工程经验与经过验证的有效性为这些模式提供了保证。
- 模式旨在重用：它们具备通用性，适合于各种问题。
- 模式定义了词汇：模式是一种定义明确的结构，因而为解决方案提供了通用的词汇。这使得大型团队在沟通时能够非常清晰地表达出各自的意图。

5.1 设计模式

在本章中，我们将介绍一些对JavaScript有意义的设计模式。但是，JavaScript的编码模式非常特殊，也是我们最感兴趣的。尽管我们要花费大量的时间和精力来领会并掌握设计模式，但理解反模式（anti-pattern）以及如何避免设计陷阱也很重要。在通常的软件开发周期中，有几个阶段会引入不良代码，这主要是在代码接近发布或代码被转交给其他维护团队的时候。如果此类不良设计被记录为反模式，就可以作为开发指南，使开发人员知道应该避开哪些陷阱，如何绕过不良的设计模式。大多数语言都有一些属于自己的反模式。基于所解决问题的种类，GOF将设计模式宽泛地划分为以下几类。

- **创建型设计模式**：该模式处理的是用于创建对象的各种机制。尽管大多数语言都提供了基本的对象创建方法，但这种模式着眼于优化的或更可控的对象创建机制。

- **结构型设计模式**：该模式考虑的是对象的组成以及对象彼此之间的关系，其意图在于将系统变化对整个对象关系所造成的影响降低到最小。
- **行为型设计模式**：该模式关注的是对象之间的依赖关系以及通信。

以下是一个识别模式分类的清单。

- 创建型模式
  - 工厂方法（Factory method）
  - 抽象工厂（Abstract factory）
  - 建造者（Builder）
  - 原型（Prototype）
  - 单例（Singleton）
- 结构型模式
  - 适配器（Adapter）
  - 桥接（Bridge）
  - 组合（Composite）
  - 装饰器（Decorator）
  - 外观（Façade）
  - 享元（Flyweight）
  - 代理（Proxy）
- 行为型模式
  - 解释器（Interpreter）
  - 模板方法（Template method）
  - 责任链（Chain of responsibility）
  - 命令（Command）
  - 迭代器（Iterator）
  - 中介者（Mediator）
  - 备忘录（Memento）
  - 观察者（Observer）
  - 状态（State）
  - 策略（Strategy）
  - 访问者（Visitor）

我们在本章中讨论的一些模式并没有出现在这个列表中，因为它们更多的是针对JavaScript的模式，或者是这些经典模式的变形。我们同样也不会去讨论那些不适合于JavaScript或者不常用的模式。

5

5.2　命名空间模式

在JavaScript中，过度使用全局作用域差不多是个禁忌。在构建大型程序时，有时候很难控制全局作用域的污染程度。命名空间能够减少程序创建的全局变量的数量，有助于避免命名冲突或过多的名称前缀。命名空间的思路是为应用程序或库创建一个全局对象，将所有的其他对象和函数全部添加到该对象中，而不是去污染全局作用域。JavaScript本身并没有与命名空间直接相关的语法，不过创建命名空间并不难。让我们来看下面的例子：

```
function Car() {}
function BMW() {}
var engines = 1;
var features = {
  seats: 6,
  airbags:6
};
```

我们将所有的东西放在了全局作用域中。这种做法就是一种反模式，绝对不是什么好主意。然而，我们可以对代码进行重构，创建单个全局对象，然后将所有的函数和对象作为该全局对象的一部分：

```
// 单一全局对象
var CARFACTORY = CARFACTORY || {};
CARFACTORY.Car = function () {};
CARFACTORY.BMW = function () {};
CARFACTORY.engines = 1;
CARFACTORY.features = {
  seats: 6,
  airbags:6
};
```

全局命名空间对象名习惯上全部都是大写。这种模式为应用程序增加了命名空间，避免你的代码和其他代码及外部库发生命名冲突。很多项目都采用公司或项目名称来创建独特的命名空间名称。

尽管这看起来是个不错的方法，既可以限制全局变量，又能够为代码添加命名空间，但多少有点繁琐；你得在每个变量和函数前面加上命名空间前缀。需要输入的内容更多，代码也变得啰嗦。另外，单个全局实例意味着任何代码都能够修改该全局实例，进而影响到其他功能——这会造成让人非常伤脑筋的副作用。在上面的例子中，有一行代码让人很好奇：`var CARFACTORY = CARFACTORY || {};`。当使用大型代码基础库的时候，你没法确定你是第一个创建该命名空间（或是命名空间属性）的人，有可能这个命名空间已经有了。为了确保仅在该命名空间不存在的前提下才进行创建，应该坚持通过短路操作符`||`来保证安全性。

5.3　模块模式

在构建大型应用程序时，你很快就会发现保持基础代码的组织化和模块化会变得越来越难。模块模式有助于维持代码清晰的独立性以及条理性。

模块可以将较大的程序分隔成较小的部分，赋予其各自的命名空间。这一点非常重要，因为一旦将代码划分成模块，这些模块就能够在其他地方重新使用。精心设计的模块接口能够使代码易于重用和扩展。

JavaScript提供了灵活的函数和对象，可以轻松地创建出健壮的模块系统。函数作用域有助于创建模块内部使用的命名空间，对象可以用来保存导出的值。

在开始研究该模式之前，让我们快速回顾一下之前学过的一些概念。

我们详细讨论过对象的字面形式，它可以按照如下形式创建键值对：

```
var basicServerConfig = {
  environment: "production",
  startupParams: {
    cacheTimeout: 30,
    locale: "en_US"
  },
  init: function () {
    console.log( "Initializing the server" );
  },
  updateStartup: function( params ) {
      this.startupParams = params;
      console.log( this.startupParams.cacheTimeout );
      console.log( this.startupParams.locale );
  }
};
basicServerConfig.init(); //"Initializing the server"
basicServerConfig.updateStartup({cacheTimeout:60, locale:"en_UK"}); //60,en_UK
```

在本例中，我们创建一个对象的字面形式，并定义了多个键值对用以创建属性和函数。

在JavaScript中，模块模式的使用率很高。模块可以帮助模拟类的概念。它使得我们能够在对象中加入公共/私有方法和变量，但最重要的是，模块能够将它们同全局作用域隔离开。变量和函数都被限制在了模块作用域中，因而也就自动避免了与使用相同名称的其他脚本产生命名冲突。

模块模式的另一个优美之处在于只暴露了公共API，其他所有与内部实现相关的细节都以私有状态保留在模块的闭包中。

与其他面向对象语言不同，JavaScript没有直接的访问修饰符，因此也就没有什么隐私（privacy）的概念，没法拥有公共变量或私有变量。我们之前讲过，在JavaScript中，可以使用函数作用域来强加这种概念。模块模式利用闭包将变量和函数的访问限制在模块内部；如果定义在

5

对象中的变量和函数被返回，那就是公共的了。

下面将之前的例子改写成模块。我们实际上使用了IIFE并返回了模块接口，也就是函数init和updateStartup：

```
var basicServerConfig = (function () {
  var environment= "production";
  startupParams= {
    cacheTimeout: 30,
    locale: "en_US"
  };
  return {
    init: function () {
      console.log( "Initializing the server" );
    },
    updateStartup: function( params ) {
      this.startupParams = params;
      console.log( this.startupParams.cacheTimeout );
      console.log( this.startupParams.locale );
    }
  };
})();
basicServerConfig.init(); //"Initializing the server"
basicServerConfig.updateStartup({cacheTimeout:60, locale:"en_UK"}); //60,en_UK
```

在本例中，basicServerConfig是全局上下文中的一个模块。要确保模块不会污染全局上下文，关键在于为模块创建命名空间。因为模块的目的在于重用，必须利用命名空间来避免名称冲突。对于basicServerConfig模块，下面的代码片段展示了如何创建命名空间：

```
// 单一全局对象
var SERVER = SERVER||{};
SERVER.basicServerConfig = (function () {
  var environment= "production";
  startupParams= {
    cacheTimeout: 30,
    locale: "en_US"
  };
  return {
    init: function () {
      console.log( "Initializing the server" );
    },
    updateStartup: function( params ) {
      this.startupParams = params;
      console.log( this.startupParams.cacheTimeout );
      console.log( this.startupParams.locale );
    }
  };
})();
SERVER.basicServerConfig.init(); //"Initializing the server"
SERVER.basicServerConfig.updateStartup({cacheTimeout:60, locale:"en_UK"});
  //60, en_UK
```

在模块中使用模块通常是个不错的主意，但并不是说模块非得使用命名空间不可。

模块模式的另一种变形用法试图克服该模式原本存在的一些问题，这种改进后的模式也被称为**暴露式模块模式**（Revealing Module Pattern，RMP）。RMP最初是由Christian Heilmann提出的，他不喜欢在调用其他函数的公共函数或是访问公共变量的时候非得使用模块名。另一个小问题是，在返回公共接口时只能采用对象的字面形式写法。考虑下面的例子：

```
var modulePattern = function(){
  var privateOne = 1;
  function privateFn(){
    console.log('privateFn called');
  }
  return {
    publicTwo: 2,
    publicFn:function(){
      modulePattern.publicFnTwo();
    },
    publicFnTwo:function(){
      privateFn();
    }
  }
}();
modulePattern.publicFn(); "privateFn called"
```

可以看到我们需要通过`publicFn()`中的`modulePattern`来调用`publicFnTwo()`。另外，公共接口是通过对象的字面形式返回的。RMP就是对这种典型的模块模式所做出的改进。其背后的主要思路就是将所有的成员都定义在私有作用域中，使用指针返回一个匿名对象，该指针所指向的私有功能需要公开暴露。

来看看怎么将上面的例子改写成RMP。这个例子主要受到了Christian个人博客的启发：

```
var revealingExample = function(){
  var privateOne = 1;
  function privateFn(){
    console.log('privateFn called');
  }
  var publicTwo = 2;
  function publicFn(){
    publicFnTwo();
  }
  function publicFnTwo(){
    privateFn();
  }
  function getCurrentState(){
    return 2;
  }
  // 通过分配共有指针来暴露私有变量
  return {
    setup:publicFn,
    count:publicTwo,
```

```
      increaseCount:publicFnTwo,
      current:getCurrentState()
    };
}();
console.log(revealingExample.current); // 2
revealingExample.setup(); // 调用privateFn
```

这里的一个重要区别就是函数和变量是在私有作用域中定义的，返回的匿名对象中带有指针，指向那些希望作为公共性质的私有变量和函数。相较于经典模块模式，这种做法更为清晰，应该优先采用。

但在产品代码中，你可能希望更多地使用标准化方法来创建模块。目前，创建模块的方法主要有两种。第一种叫作CommonJS模块。CommonJS模块通常更适合于服务器端JavaScript环境（如Node.js）。CommonJS模块中包含了一个`require()`函数，能够接受模块名并返回模块接口。这种形式是由CommonJS志愿者小组提出的，该组织的目标是设计、提出原型并标准化JavaScript API。CommonJS模块由两部分组成。一部分是模块要暴露出的变量和函数列表；如果你将变量或函数赋值给`module.exports`，就会将其暴露给外部。另一部分是`require()`函数，模块可以用它导入其他模块导出的内容：

```
// 添加依赖模块
var crypto = require('crypto');
function randomString(length, chars) {
  var randomBytes = crypto.randomBytes(length);
  ...
  ...
}
// 导出该模块，使其可供其他模块使用
module.exports=randomString;
```

CommonJS模块在服务器端由Node.js支持，在浏览器端由curl.js支持。

另一种JavaScript模块形式叫作**异步模块定义**（Asynchronous Module Definition，AMD）。这是一种浏览器优先（browser-first）的模块，能够支持异步行为。AMD使用`define()`函数来定义模块，该函数接受一个包含了模块名称的数组和一个回调函数。模块载入后，`define()`就会执行这个回调函数，并将已载入的模块接口作为参数。AMD的目的在于以异步方式载入模块及其依赖关系。`define()`函数依据以下签名（signature）来定义命名或未命名模块：

```
define(
  module_id /*可选*/,
  [dependencies] /*可选*/,
  definition function /*用于实例化模块或对象的函数*/
);
```

可以添加一个不具有依赖性的模块：

```
define(
{
```

```
  add: function(x, y){
    return x + y;
  }
});
```

下面的代码片段展示了一个模块，该模块依赖另外两个模块：

```
define( "math",
  // 依赖于这两个模块
  ["sum", "multiply"],
  // 模块定义函数
  // 将依赖关系（foo和bar）映射为函数参数
  function ( sum, multiply ) {
    // 返回的值决定了模块的导出
    // （也就是我们希望暴露的功能）

    // 在这里创建你自己的模块
    var math = {
      demo : function () {
        console.log(sum.calculate(1,2));
        console.log(multiply.calculate(1,2));
      }
    };
  return math;
});
```

`require`模块的用法如下：

```
require(["math","draw"], function ( math,draw ) {
  draw.2DRender(math.pi);
});
```

RequireJS（http://requirejs.org/docs/whyamd.html）是实现了AMD的模块装载器之一。

ES6 模块

两个独立的模块系统加上不同的模块装载器着实有点令人生畏，ES6尝试解决这个问题。ES6提出了一个模块规范，试图保留CommonJS和ADM模块模式的优点。ES6模块的语法类似于CommonJS，另外还支持异步装载以及可配置的模块装载：

```
//json_processor.js
function processJSON(url) {
  ...
}
export function getSiteContent(url) {
  return processJSON(url);
}
//main.js
import { getSiteContent } from "json_processor.js";
content=getSiteContent("http://google.com/");
```

ES6的导出功能可以让你使用类似于CommonJS的方式导出函数或变量。在需要使用导入函数的代码中，可以用关键字import来指定从哪里导入所需的依赖。所需的依赖导入后，就可以作为程序的组成部分使用了。随后我们会讨论如何在不支持ES6的环境中使用ES6。

5.4　工厂模式

工厂模式是另一种流行的对象创建模式，它并不需要使用构造函数。该模式提供了一个用于创建对象的接口。根据传入工厂的类型，可以创建出特定类型的对象。这种模式常见的实现通常是利用类或类的静态方法。这样的类或方法的目的如下：

- 在创建相似对象时，抽象出重复出现的操作
- 允许工厂的用户在无需了解对象创建的内部细节的情况下创建对象

让我们用一个常见的例子来理解工厂模式的用法。假设我们有：

- 一个构造函数CarFactory()
- CarFactory中一个叫作make()的静态方法，该方法知道如何创建car类型的对象
- 特定的car类型，例如CarFactory.SUV、CarFactory.Sedan等

我们希望像下面这样使用CarFactory：

```
var golf = CarFactory.make('Compact');
var vento = CarFactory.make('Sedan');
var touareg = CarFactory.make('SUV');
```

这里给出了这种工厂的实现方法。下面的实现非常标准。我们用编程的方式调用了构造函数，创建指定类型的对象——CarFactory[const].prototype = new CarFactory();。

我们将对象类型映射为构造函数。该模式的实现方法不止一种：

```
// 工厂构造函数
function CarFactory() {}
CarFactory.prototype.info = function() {
  console.log("This car has "+this.doors+" doors and a
    "+this.engine_capacity+" liter engine");
};
// 静态工厂方法
CarFactory.make = function (type) {
  var constr 0= type;
  var car;
  CarFactory[constr].prototype = new CarFactory();
  // 创建新的实例
  car = new CarFactory[constr]();
  return car;
};

CarFactory.Compact = function () {
```

```
  this.doors = 4;
  this.engine_capacity = 2;
};
CarFactory.Sedan = function () {
  this.doors = 2;
  this.engine_capacity = 2;
};
CarFactory.SUV = function () {
  this.doors = 4;
  this.engine_capacity = 6;
};
var golf = CarFactory.make('Compact');
var vento = CarFactory.make('Sedan');
var touareg = CarFactory.make('SUV');
golf.info(); //"This car has 4 doors and a 2 liter engine"
```

建议在JS Bin中实践这个例子，通过编写代码来理解这些概念。

5.5 mixin 模式

mixin模式能够显著减少代码中重复出现的功能，有助于功能重用。我们可以将能够共享的功能放到mixin中，以此降低共享行为的重复数量。这样你就可以将注意力放在构建实际功能上，而不是一再去重复那些可以共享的行为。来考虑下面的例子。我们想创建一个定制的日志记录器（logger），任何对象实例都可以使用。这个日志记录器会在希望使用/扩展mixin的对象之间共享：


```
var _ =require('underscore');
// 将共享的功能封装进CustomLogger
var logger = (function () {
  var CustomLogger = {
    log: function (message) {
      console.log(message);
    }
  };
  return CustomLogger;
}());

// 需要定制的日志记录器来记录系统特定日志的对象
var Server = (function (Logger) {
  var CustomServer = function () {
    this.init = function () {
      this.log("Initializing Server…");
    };
  };

  // 将CustomLogger的成员复制/扩展为CustomServer
  _.extend(CustomServer.prototype, Logger);
  return CustomServer;
}(logger));

(new Server()).init(); // 初始化服务器……
```

在本例中，我们使用了Underscore.js中的_.extend——上一章中讲过该函数。这个函数用来将源对象（Logger）中的所有属性都复制到目标对象（CustomServer.prototype）中。正如在本例中所看到的，我们创建了一个共享的CustomLogger对象，供需要该功能的对象实例使用。例如CustomServer对象——在它的init()方法中就调用了日志记录器的log()方法。CustomServer之所以能使用这个方法，是因为我们利用Underscore的extend()扩展了CustomLogger。我们动态地将mixin的功能加入到了对象中。重要的是要理解mixin和继承之间的区别。如果有可以在多个对象和类层次之间共享的功能，可以使用mixin；如果要共享的功能是在单个类层次中，可以使用继承。在原型继承中，如果继承来自原型，那么对原型做出的修改会影响到从原型中继承的一切内容。如果不希望出现这种情况，可以使用mixin。

5.6　装饰器模式

装饰器模式的主要思想是使用一个空白对象（plain object）展开设计，该对象有一些基本的功能。随着设计的深入，可以使用现有的装饰器来增强该空白对象，这是一种在面向对象领域中非常流行的模式，尤其是在Java中。举一个BasicServer的例子——这是一个具有最基础功能的服务器。可以对这些基础功能进行装饰以满足特定的需求。这个服务器需要在不同的端口服务于PHP和Node.js。各种不同的功能都可以装饰到基础服务器上：

```
var phpServer = new BasicServer();
phpServer = phpServer.decorate('reverseProxy');
phpServer = phpServer.decorate('servePHP');
phpServer = phpServer.decorate('80');
phpServer = phpServer.decorate('serveStaticAssets');
phpServer.init();
```

Node.js服务器需要执行以下操作：

```
var nodeServer = new BasicServer();
nodeServer = nodeServer.decorate('serveNode');
nodeServer = nodeServer.decorate('3000');
nodeServer.init();
```

在JavaScript中，实现装饰器模式的方法不止一种。我们将讨论一种通过列表实现的方法，这种实现不依赖于继承或方法调用链：

```
// 实现最小化的BasicServer
function BasicServer() {
  this.pid = 1;
  console.log("Initializing basic Server");
  this.decorators_list = []; // 空的装饰器列表
}
// 列出所有的装饰器
BasicServer.decorators = {};

// 将每个装饰器添加到BasicServer的装饰器列表中
// 列表中的每个装饰器都会被应用到BasicServer实例
```

```
BasicServer.decorators.reverseProxy = {
  init: function(pid) {
    console.log("Started Reverse Proxy");
    return pid + 1;
  }
};
BasicServer.decorators.servePHP = {
  init: function(pid) {
    console.log("Started serving PHP");
    return pid + 1;
  }
};
BasicServer.decorators.serveNode = {
  init: function(pid) {
    console.log("Started serving Node");
    return pid + 1;
  }
};
// 每次调用decorate()时，都将装饰器推入列表
BasicServer.prototype.decorate = function(decorator) {
  this.decorators_list.push(decorator);
};
// init()方法遍历所有应用于BasicServer的装饰器，
// 在所有的装饰器上执行init()方法
BasicServer.prototype.init = function () {
  var running_processes = 0;
  var pid = this.pid;
  for (i = 0; i < this.decorators_list.length; i += 1) {
    decorator_name = this.decorators_list[i];
    running_processes = BasicServer.decorators[decorator_name].init(pid);
  }
  return running_processes;
};

// 创建提供PHP服务的服务器
var phpServer = new BasicServer();
phpServer.decorate('reverseProxy');
phpServer.decorate('servePHP');
total_processes = phpServer.init();
console.log(total_processes);

// 创建提供Node服务的服务器
var nodeServer = new BasicServer();
nodeServer.decorate('serveNode');
nodeServer.init();
total_processes = phpServer.init();
console.log(total_processes);
```

做实质工作的是`BasicServer.decorate()`和`BasicServer.init()`。我们将所有用到的装饰器都推入`BasicServer`的装饰器列表中。在`int()`中，我们从这个装饰器列表中执行或应用每个装饰器的`init()`方法。这种做法不需要使用继承，更为简洁。Stoyan Stefanov在《JavaScript模式》一书中讲述过该方法，由于其简单性，成为了JavaScript开发人员中的主流方法。

5.7　观察者模式

我们先来看一下观察者模式与语言无关的定义。在GOF的《设计模式》一书中，是这样定义观察者模式的：

> 对目标状态感兴趣的一个或多个观察者，通过将自身与该目标关联在一起的形式进行注册。当目标出现观察者可能感兴趣的变化时，发出提醒消息，进而调用每个观察者的更新方法。如果观察者对目标状态不再感兴趣，只需要解除关联即可。

在观察者模式中，目标保存了一个对其依赖的对象列表（称为观察者），并在自身状态发生变化时通知这些观察者。目标所采用的通知方式是广播。观察者如果不想再被提醒，可以把自己从列表中移除。理解了这些，我们可以对该模式中的参与者做出如下定义。

- **目标**（Subject）：保存观察者列表，拥有可以用来添加、删除和更新观察者的方法。
- **观察者**（Observer）：为那些需要在目标状态发生变化时得到提醒的对象提供接口。

让我们来创建一个能够添加、删除和提醒观察者的目标：

```
var Subject = ( function( ) {
  function Subject() {
    this.observer_list = [];
  }
  // 该方法用于向内部列表中添加观察者
  Subject.prototype.add_observer = function ( obj ) {
    console.log( 'Added observer' );
    this.observer_list.push( obj );
  };
  Subject.prototype.remove_observer = function ( obj ) {
    for( var i = 0; i < this.observer_list.length; i++ ) {
      if( this.observer_list[ i ] === obj ) {
        this.observer_list.splice( i, 1 );
        console.log( 'Removed Observer' );
      }
    }
  };
  Subject.prototype.notify = function () {
    var args = Array.prototype.slice.call( arguments, 0 );
    for( var i = 0; i<this.observer_list.length; i++ ) {
      this.observer_list[i].update(args);
    }
  };
  return Subject;
})();
```

`Subject`的实现方法非常直观。`notify()`方法的重要之处在于调用所有观察者对象的`update()`方法来广播更新。

现在来定义一个能够创建随机推文的简单对象，该对象提供了一个接口，可以使用addObserver()和removeObserver()方法来添加和删除观察者。它也可以使用新获取的推文来调用Subject的notify()方法。如果出现这种情况，所有的观察者都会将新发布的推文作为参数，发出有推文更新的广播：

```
function Tweeter() {
  var subject = new Subject();
  this.addObserver = function ( observer ) {
    subject.add_observer( observer );
  };
  this.removeObserver = function (observer) {
    subject.remove_observer(observer);
  };
  this.fetchTweets = function fetchTweets() {
    // tweet
    var tweet = {
      tweet: "This is one nice observer"
    };
    // 提示观察者股票发生的变化
    subject.notify( tweet );
  };
}
```

添加两名观察者：

```
var TweetUpdater = {
  update : function() {
    console.log( 'Updated Tweet - ', arguments );
  }
};
var TweetFollower = {
  update : function() {
    console.log( '"Following this tweet - ', arguments );
  }
};
```

两名观察者都有update()方法，该方法将由Subject.notify()方法调用。现在我们就可以通过Tweeter的接口将观察者添加到Subject中了：

```
var tweetApp = new Tweeter();
tweetApp.addObserver( TweetUpdater );
tweetApp.addObserver( TweetFollower );
tweetApp.fetchTweets();
tweetApp.removeObserver(TweetUpdater);
tweetApp.removeObserver(TweetFollower);
```

产生的输出信息如下：

```
Added observer
Added observer
Updated Tweet - { '0': [ { tweet: 'This is one nice observer' } ] }
```

```
"Following this tweet - { '0': [ { tweet: 'This is one nice observer' } ] }
Removed Observer
Removed Observer
```

这就是一个用来演示观察者模式概念的基本实现。

5.8　JavaScript 的 Model-View-*模式

模型–视图–控制器（Model-View-Controller，MVC）、**模型–视图–表现器**（Model-View-Presenter，MVP）、**模型–视图–视图模型**（Model-View-ViewModel，MVVM）流行于服务器应用，但是近年来，JavaScript应用也开始使用这些模式来构架和管理大型项目。涌现出的很多JavaScript框架都支持MV*模式。我们将使用Backbone.js来讨论几个例子。

5.8.1　模型–视图–控制器

MVC是一种流行的结构型模式，其思路是将应用程序划分成三部分，从而将信息的内部描述与表现层分离开。MVC是由多个组件构成的。模型是应用程序对象，视图是底层模型对象的表现，控制器是根据用户交互处理用户界面的行为方式。

5.8.2　模型

模型是用来描述应用程序数据的构件。它们与用户界面和路由逻辑无关。模型上的变化通常会按照观察者设计模式通知到视图层。模型中也可以包含用于验证、创建或删除数据的代码。在数据发生改变时，自动提醒视图进行响应的能力，使得框架（如Backbone.js、Amber.js等）在构建MV*应用中发挥重大作用。下面的例子展示了一个典型的Backbone模型：

```
var EmployeeModel = Backbone.Model.extend({
  url: '/employee/1',
  defaults: {
    id: 1,
    name: 'John Doe',
    occupation: null
  }
  initialize: function() {
  }
});
var JohnDoe = new EmployeeModel();
```

模型结构在不同的框架中可能会有差别，但通常都具有一些共性。在大多数实际的应用中，模型都会被放入内存或数据库中以持久化。

5.8.3 视图

视图是模型的可视化表现。通常，模型的状态在递交给视图层之前需要处理、过滤或修整。在JavaScript中，视图负责渲染和操作DOM元素。视图观察模型，在模型出现变化时得到提醒。在用户与视图交互时，模型的某些属性可以通过视图层修改（通常是通过控制器）。在JavaScript框架（如Backbone）中，创建视图的是模板引擎，如Handlebar.js（http://handlebarsjs.com/）或mustache.js（https://mustache.github.io/）。这些模板本身并不是视图，它们会观察模型并根据模型的变化更新视图。来看一个在Handlebar中定义的视图：

```
<li class="employee_photo">
  <h2>{{title}}</h2>
  <img class="emp_headshot_small" src="{{src}}"/>
  <div class="employee_details">
    {{employee_details}}
  </div>
</li>
```

上例的视图中包含了标签，标签中含有模板变量，这些变量通过特定的语法来分隔。例如在Handlebar.js中，模板变量使用`{{}}`来界定。框架传送数据时通常使用JSON格式。框架负责从模型中生成视图，具体的细节无需用户关心。

5.8.4 控制器

控制器作用于模型和视图之间，负责在用户修改视图属性时更新模型。大多数JavaScript框架并没有遵循控制器的典型定义。比如说，Backbone就没有控制器的概念，只有一个叫作**路由器**（router）的东西，负责处理路由逻辑。可以把控制器看作视图和路由器的组合，因为很多同步模型和视图的逻辑都是由视图本身来完成的。一个典型的Backbone路由器如下所示：

```
var EmployeeRouter = Backbone.Router.extend({
  routes: { "employee/:id": "route" },
  route: function( id ) {
    ...view render logic...
  }
});
```

5.9 模型–视图–表现器

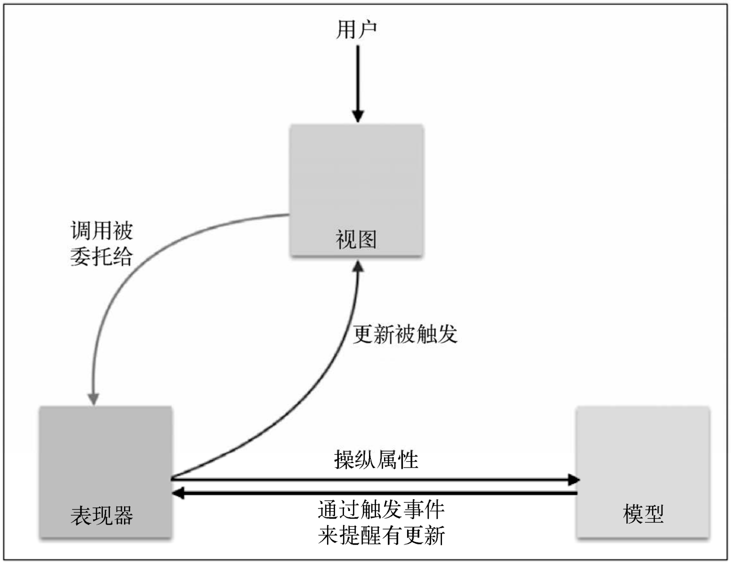
模型–视图–表现器是之前讨论过的原始MVC模式的一种变形。MVC和MVP的目的都是分离，但两者在基本层面上有诸多不同。MVP中的表现器是视图必不可少的逻辑，视图中的所有调用都被委托给表现器。表现器也会观察模型，在模型出现变化时更新视图。很多作者采用视图是因为表现器不仅将模型与视图绑定在了一起，而且它还能够扮演传统控制器的角色。MVP有各种实现，还没有框架能够提供直接可用的MVP模式。以下是MVP与MVC在实现上的主要差异：

- 视图不引用模型
- 表现器引用模型，负责在模型发生变化时更新视图

MVP一般有两种实现方式。

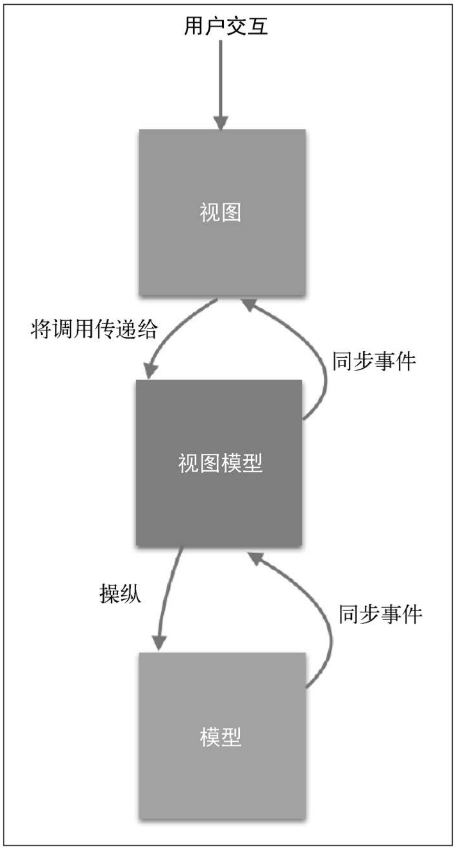
- 被动视图（passive view）：视图尽可能简单，所有的业务逻辑都放在表现器中。例如，一个空白的Handlebars模板就可以视为被动视图。
- 监视控制器（supervising controller）：视图包含大部分的声明式逻辑（declarative logic）。当视图中简单的声明式逻辑不够用的时候，由表现器接管。

下图描述了MVP的架构：



5.10　模型−视图−视图模型

MVVM最初是由微软发明并应用于Windows Presentation Foundation（WPF）和Silverlight。作为MVC和MVP的一种变形，MVVM进一步尝试将用户界面（视图）与业务模型及应用程序行为分离。除了我们在MVC和MVP中讨论过的领域模型（domain model），MVVM还创建了一个新的模型层，这个模型层添加了各种属性作为视图层的接口。假设我们在用户界面上有一个复选框，`IsChecked`属性用来保存复选框的状态。在MVP中，视图拥有这个属性，表现器能够设置这个属性。但是在MVVM中，拥有`IsChecked`属性的是表现器，视图负责与其同步。这时的表现器所做的已经不是典型表现器所干的活了，它现在叫作视图模型：



这些方法的实现细节依赖于我们所要解决的问题以及所使用的框架。

5.11　小结

在构建大型应用程序时，我们发现某些问题模式反复地出现。这类问题都有明确的应对方法，可以拿来重用以构建一套健壮的解决方案。在本章中，我们讨论了一些重要的模式及其设计理念。大多数现代JavaScript应用都采用了这些模式。极少有哪个大规模的系统构建没有实现模块、装饰器、工厂或者MV*模式，这些都是本章中讲到的基本思想。在下一章中，我们将学习各种测试及调试技术。

第 6 章

测试与调试

6

在编写JavaScript应用的时候，你很快就会发现一套全面详实的测试策略是必不可少的。事实上，不编写足够的测试几乎总是一个坏主意。测试必须能够覆盖代码所有重要的功能，以确保以下要点：

- 现有代码的行为和规范中的一致
- 新代码不会违背规范所定义的行为

这两点非常重要。很多工程师认为只有第一点才是使用足够的测试覆盖代码的唯一原因。测试覆盖面的最显著优势在于它能够确保推送到生产系统的代码几乎是没有错误的。编写测试用例来巧妙地覆盖最广的代码功能区域，通常能够很好地反映出代码的整体质量。在这一点上，不应该有异议或是妥协。遗憾的是，很多产品系统仍然没有足够的代码覆盖面。营造出一种工程文化，使开发人员认同编写测试与编写代码同等重要，这一点不可小视。

第二点更重要。遗留系统（legacy system）通常很难管理。在处理由他人或大型分散式团队编写的代码时，很容易引入错误，把事情搞砸。就算是最优秀的工程师也会犯错。在使用不熟悉的大型代码基础库的时候，如果不借助于全面的测试覆盖，错误在所难免。由于对做出的代码变更没有自信（因为没有测试用例来验证变更），你发布的代码自然根基不牢、速度迟缓、存在着各种隐藏的错误。

你不愿意重构或优化代码，是因为不确定对代码基础库做出的改动会不会捅篓子（还是那句话，没用测试用例来验证你所做出的改动）——这就形成了一个恶性循环。如同一位土木工程师所言："虽然我建起了这座桥梁，但对它的质量可没什么信心。它可能立马就会塌掉，也可能永远都不会。"尽管听起来可能有些夸张，但我的确见过不少颇具影响力的产品代码没进行测试覆盖就被推送出来了。这就是在冒险，应该避免。如果编写了足够的测试用例去覆盖大部分的功能代码，那么在对这些代码做出修改的时候，你立刻就能知道这次新的改动有没有问题。如果改动没有通过测试用例，那就是有问题了。如果重构和测试场景不符，你也能知道是哪的问题——所有这些在代码被推送成产品之前就进行了。

近些年，如测试驱动开发、自测试代码（self-testing code）这样的理念日渐兴起，尤其是敏

捷方法论。这些思想基本上都有其合理之处，有助于你写出高质量的代码——对其充满信心的代码。这里所提及的概念都会在本章中讲到。你将了解如何在现代JavaScript中编写良好的测试用例，还将学习一些代码调试的工具和方法。由于缺乏相应的工具，JavaScript通常不太好进行测试及调试，不过如今的现代化工具使得这一过程变得既简单又自然。

6.1 单元测试

当我们说到测试用例的时候，主要指的是**单元测试**。认为测试单元只能是函数的想法是不对的，单元（或工作单元）是一个由单一行为组成的逻辑单元。单元应该能够通过一个公共接口调用，能够独立进行测试。

因此，一个单元测试具有下列功能：

- 测试单个逻辑功能
- 不依照指定的执行顺序运行
- 负责自身的依赖关系以及模拟数据（mock data）
- 对于相同的输入，总是返回相同的结果
- 一目了然，可维护，具有良好的可读性

> Martin Fowler提倡采用**测试金字塔**（http://martinfowler.com/bliki/TestPyramid.html）策略，以确保使用大量的单元测试来保证最大程度的代码覆盖面。所谓的测试金字塔是说你编写的低层单元测试数量应该比高层的集成和UI测试多得多。

本章会讨论两种重要的测试策略。

6.1.1 测试驱动开发

测试驱动开发（test-driven development，TDD）近几年来声名卓越。这个概念最初是作为**极限编程**方法论的一部分提出的，其思想是采用短期迭代开发，在开发周期中先写测试用例。每个周期过程如下。

(1) 添加一个测试用例作为特定代码单元的规范。

(2) 运行现有的测试用例组，检查所编写的新测试用例是否没能通过——应该通过不了（因为还没有单元代码）。这一步确保了当前测试一切正常。

(3) 编写代码，主要作为验证测试用例之用。这些代码无需优化、重构，甚至不用完全正确。在现阶段，这种情况很正常。

(4) 重新进行测试，检查所有测试用例是否通过。在这一步之后，你就能够确信新代码没问题了。

(5) 重构代码，确保优化单元代码并处理所有的极端情况。

上述步骤重复应用于所添加的新代码。这是一种很好的策略，非常适合敏捷方法。只有在可测试的代码单元保持短小并且只针对测试用例的时候，TDD才可行。编写短小、模块化以及精准的代码单元，并且具备能够用于验证测试用例的输入和输出，这才是关键所在。

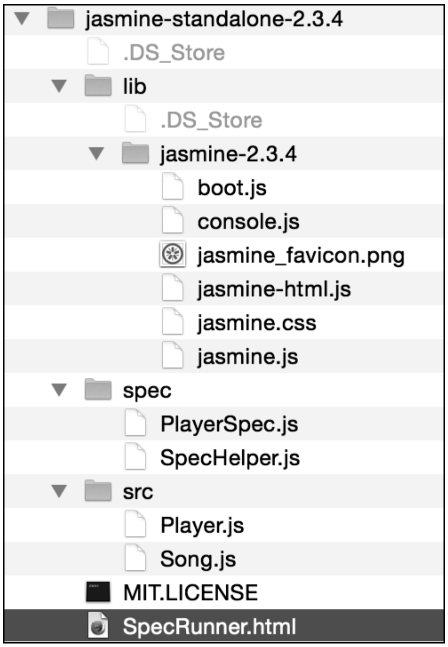
6.1.2　行为驱动开发

在尝试实施TDD时，一个很常见的问题就是用词和**正确性**（correctness）的定义。行为驱动开发（behavior-driven development，BDD）试图在编写测试用例时引入一种**通用语言**（ubiquitous language），这种语言能够确保业务团队和工程团队在表达上取得一致。

我们打算使用Jasmine作为主要的BDD框架，探究不同的测试策略。

> 你可以从https://github.com/jasmine/jasmine/releases/download/v2.3.4/jasmine-standalone-2.3.4.zip下载独立的安装包进行安装。

解压缩安装包之后，会看到如下目录结构：



lib目录包含了项目中用于编写Jasmine测试用例所需的JavaScript文件。打开SpecRunner.html，会看到包含了下列JavaScript文件：

```
<script src="lib/jasmine-2.3.4/jasmine.js"></script>
<script src="lib/jasmine-2.3.4/jasmine-html.js"></script>
<script src="lib/jasmine-2.3.4/boot.js"></script>

<!-- include source files here… -->
<script src="src/Player.js"></script>
<script src="src/Song.js"></script>
<!-- include spec files here… -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
```

前三个是Jasmine自身的框架文件，余下的包括待测试的源文件以及实际的测试规范。

接下来用个很普通的例子来实验一下Jasmine。创建名为bigfatjavascriptcode.js的文件，将其放入src/目录。我们将测试以下函数：

```
function capitalizeName(name){
  return name.toUpperCase();
}
```

这是个很简单的函数，它只做一件事。它接收一个字符串，然后返回该字符串的大写形式。我们将围绕该函数测试各种场景。这就是之前说过的代码单元。

接下来要创建测试规范。新建一个名为test.spec.js的JavaScript文件，将其放入spec/目录。你要把下面两行代码添加到SpecRunner.html中：

```
<script src="src/bigfatjavascriptcode.js"></script>
<script src="spec/test.spec.js"></script>
```

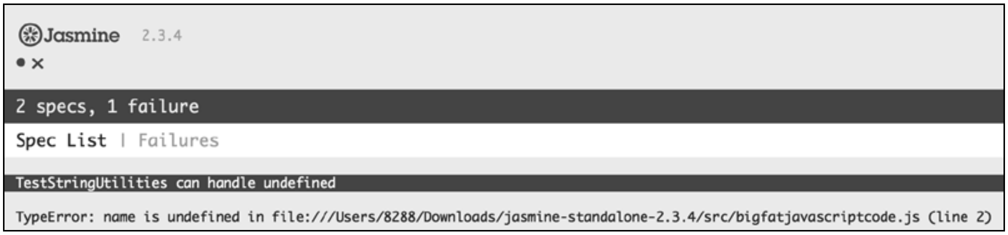
包含文件出现的先后次序并不重要。当运行SpecRunner.html时，会看到如下内容：



这是Jasmine的报告，显示了所执行的测试数量以及测试失败/成功的次数。现在，让我们来想办法使测试用例失败。下面要测试的是当一个未定义变量被传入函数时会出现什么情况。再添加另一个测试用例：

```
it("can handle undefined", function() {
  var str= undefined;
  expect(capitalizeName(str)).toEqual(undefined);
});
```

这次运行SpecRunner.html时会看到下面的结果：



如你所见，此次测试用例中所出现的问题以错误栈的形式详细地显示了出来。现在，我们来修复这些错误。在最初的JavaScript代码中，我们可以这样处理未定义变量的问题：

```
function capitalizeName(name){
  if(name){
    return name.toUpperCase();
  }
}
```

修改过之后，测试用例就能通过了，你会看到如下的Jasmine报告：



这与测试驱动开发很形似。编写测试用例，加入必要的代码来验证规范，然后重新运行测试套件。下面让我们来弄清楚Jasmine测试的结构。

我们的测试规范如下：

```
describe("TestStringUtilities", function() {
  it("converts to capital", function() {
    var str = "albert";
    expect(capitalizeName(str)).toEqual("ALBERT");
  });
  it("can handle undefined", function() {
    var str= undefined;
    expect(capitalizeName(str)).toEqual(undefined);
  });
});
```

describe("TestStringUtilities"是一组测试套件。测试套件的名字应该能够描述所进

行测试的代码单元——这可以是一个函数或是一组相关的功能。在规范中，调用了Jasmine的全局函数it()，你需要将规范标题以及测试用例所使用的测试函数传给它。这个函数就是实际的测试用例。你可以用expect()函数捕获断言（assertion）或是一般的expectation。如果所有的expectation均为true，则规范通过。任何有效的JavaScript代码都能够写到descrie()和it()函数中。对需要作为expectation的一部分进行验证的值，可以使用匹配器（matcher）进行匹配。在本例中，toEqual()是用来对两个值做等量匹配的匹配器。Jasmine包含了大量的匹配器，能够适用于绝大部分常见用例。Jasmine支持的一些常见匹配器如下。

- toBe()：检查所比较的两个对象是否相同。其效果和===一样，如以下代码所示：

```
var a = { value: 1};
var b = { value: 1};

expect(a).toEqual(b); // 成功，等同于==
expect(b).toBe(b);    // 失败，等同于===
expect(a).toBe(a);    // 成功，等同于===
```

- not：使用not前缀否定某个匹配器。例如，expect(1).not.toEqual(2);会否定toEqual()所做出的匹配。
- toContain()：检查某个元素是否存在于数组中。这并不是toBe()那样的严格对象匹配。例如下面的代码：

```
expect([1, 2, 3]).toContain(3);
expect("astronomy is a science").toContain("science");
```

- toBeDefined()和toBeUndefined()：可以很方便地检查变量是否定义过。
- toBeNull()：检查变量值是否为null。
- toBeGreaterThan()和toBeLessThan()：这两个匹配器执行数字比较（也可以用于字符串）。

```
expect(2).toBeGreaterThan(1);
expect(1).toBeLessThan(2);
expect("a").toBeLessThan("b");
```

spy是Jasmine一个值得注意的特性。在编写大型系统时，不可能保证所有的系统总是可用、不出差错，但同时又不希望由于依赖性故障而使单元测试无法通过。为了模仿出一个待测试的代码单元所需的全部依赖都可用的情形，我们模拟这些疑虑，给出所期望的响应。模拟（mocking）是测试很重要的一方面，大多数测试框架都支持模拟。Jasmine可以使用spy特性实现模拟。实际上，它是对那些在编写测试用例时我们尚未准备好，但又担负着部分功能的函数进行了插桩（stub），我们需要跟踪这些依赖，不能忽略。考虑下面的例子：

```
describe("mocking configurator", function() {
  var configurator = null;
  var responseJSON = {};
```

```
    beforeEach(function() {
      configurator = {
        submitPOSTRequest: function(payload) {
        // 这是一个模拟服务，最终会被真实服务替代
        console.log(payload);
        return {"status": "200"};
        }
      };
      spyOn(configurator, 'submitPOSTRequest').and.returnValue({"status": "200"});
      configurator.submitPOSTRequest({
        "port":"8000",
        "client-encoding":"UTF-8"
      });
    });

    it("the spy was called", function() {
      expect(configurator.submitPOSTRequest).toHaveBeenCalled();
    });

    it("the arguments of the spy's call are tracked", function() {
      expect(configurator.submitPOSTRequest).toHaveBeenCalledWith({"port":"8000",
  "client-encoding":"UTF-8"});
    });
  });
```

我们在编写测试用例的时候还没有所依赖的`configurator.submitPOSTRequest()`，可能是尚未实现，也可能是有人正在对其进行修复。不管怎么说，现在都用不了。为了能够完成测试，我们需要模拟该依赖。Jasmine的spy可以用模拟来替换这个函数，并跟踪其执行。

在这种情况下，我们需要确保调用了指定的依赖。当依赖实现好之后，我们会再回到该测试用例，检查它是否符合规范，但目前，所需做的就是保证依赖被调用。Jasmine的`tohaveBeenCalled()`函数可以跟踪函数的执行过程，即便这个函数只是模拟的。我们可以使用`tohaveBeenCalledWith()`判断插桩函数在调用时是否使用了正确的参数。在其他一些有趣的场景中也可以使用spy。但受限于本章的篇幅，我们无法为你逐一道来，但我鼓励你自己去发现这些领域。

> 可以参考Jasmine的用户手册以获取关于Jasmine spy更多的相关信息：https://jasmine.github.io/2.0/introduction.html。

> **Mocha、Chai和Sinon**
>
> 尽管Jasmine是名声最大的JavaScript测试框架，Mocha和Chai在Node.js环境中也逐渐崭露头角。Mocha是一种可用于描述并运行测试用例的测试框架。Chai是由Mocha支持的断言库（assertion library）。Sinon.js在创建测试模拟和插桩方面得心应手。我们不会在本书中讨论这些框架，不过有了Jasmine的使用经验，学习这些框架会更容易。

6.2 JavaScript 调试

如果你不是一名彻头彻尾的程序员新手，我相信你肯定花费时间调试过自己或别人的代码。调试几乎形同艺术。每种语言都有不同的调试方法和难题。在传统上，JavaScript是一门难以调试的语言。我曾经耗费了几天的时间，试图用`alert()`函数调试那些糟糕的JavaScript代码。好在像Mozilla Firefox和Google Chrome这样的现代浏览器已经配备了非常出色的开发者工具，帮助在浏览器中调试JavaScript。像IntelliJ和WebStorm这样的IDE，也对JavaScript及Node.js调试提供了很好的支持。在本章中，我们主要关注的是Google Chrome内置的开发者工具。FireFox支持Firebug扩展，同时也内置了优秀的开发者工具，但用法多少和Google Chrome的Developer Tools（DevTools）类似，我们将讨论能够用于这两种工具的通用调试方法。

在讨论特定的调试技术之前，让我们先了解一下在调试代码时感兴趣的那些错误类型。

6.2.1 语法错误

如果代码出现不符合JavaScript语言语法的地方，解释器会拒绝执行这部分代码。如果你用的IDE有语法检测功能的话，这些错误很容易发现。大多数现代IDE都能够处理这种错误。之前我们讨论过一些有用的工具（如JSLint和JSHint）都能够捕获代码中的语法问题，它们能够分析代码并标出语法错误。JSHint的输出一目了然。例如，下面的输出显示出了代码中大量需要修改的地方。这段代码取自我现有的一个项目：

```
temp git:(dev_branch) ✗ jshint test.js
test.js: line 1, col 1, Use the function form of "use strict".
test.js: line 4, col 1, 'destructuring expression' is available in ES6 (use esnext
option) or Mozilla JS extensions (use moz).
test.js: line 44, col 70, 'arrow function syntax (=>)' is only available in ES6 (use
esnext option).
test.js: line 61, col 33, 'arrow function syntax (=>)' is only available in ES6 (use
esnext option).
test.js: line 200, col 29, Expected ')' to match '(' from line 200 and instead saw ':'.
test.js: line 200, col 29, 'function closure expressions' is only available in Mozilla
JavaScript extensions (use moz option).
test.js: line 200, col 37, Expected '}' to match '{' from line 36 and instead saw ')'.
test.js: line 200, col 39, Expected ')' and instead saw '{'.
test.js: line 200, col 40, Missing semicolon.
```

6

6.2.2 使用严格模式

我们之前简要讨论过**严格模式**。JavaScript的严格模式能够标出或消除JavaScript的一些静默错误（silent error）。严格模式能够为这些毛病抛出错误，而不是悄无声息地运行失败。严格模式也有助于将失误（mistake）变成实际的错误（error）。有两种强制执行严格模式的方法。如果希望将整个脚本置入严格模式，可以在你的JavaScript程序的第一行写上`use strict`语句。如果希

望将严格模式限制在某个函数范围内，可以在函数的第一行写上这句指令：

```
function strictFn(){
// 该行使得其下的所有代码都处于严格模式
'use strict';
…
function nestedStrictFn() {
// 该函数内的代码也处于严格模式
…
}
}
```

6.2.3　运行时异常

当你在执行代码并试图引用一个未定义的变量或是处理null时，就会出现错误。如果发生运行时异常，引发异常的那行代码之后的所有代码都不会再执行。必须能够正确地处理这样的异常情景。异常处理不仅能够帮助避免崩溃，而且也有助于调试。你可以把有可能出现运行时异常的代码放到`try{}`块中。如果块中的代码产生了运行时异常，对应的处理程序就能够捕获到。处理程序定义在`catch(exception){}`块中。来看下面的例子：

```
try {
  var a = doesnotexist; // 抛出运行时异常
} catch(e) {
  console.log(e.message); // 处理异常
  //打印出"doesnotexist is not defined"
}
```

在这个例子中，`var a = doesnotexist;`打算将一个未定义的变量`doesnotexist`赋给另一个变量`a`。这会导致运行时异常。如果我们把这段有问题的代码放入`try{} catch(){}`块中，当异常发生（或抛出）时，执行过程会在`try{}`块处停止，直接进入`catch(){}`中的处理程序。`catch`处理程序负责处理异常情况。本例中，我们在控制台中显示用于调试的错误信息。你可以明确地抛出异常来触发代码中未处理的情形。考虑以下代码：

```
function engageGear(gear){
  if(gear==="R"){ console.log ("Reversing");}
  if(gear==="D"){ console.log ("Driving");}
  if(gear==="N"){ console.log ("Neutral/Parking");}
  throw new Error("Invalid Gear State");
}
try
{
  engageGear("R"); //Reversing
  engageGear("P"); //Invalid Gear State
}
catch(e){
  console.log(e.message);
}
```

本例中，对有效的换挡状态（R、N、D）做了处理，但如果接收到一个无效状态，我们会明确地抛出一个异常，清晰地说明原因。在调用有可能会抛出异常的函数时，我们会把对应的代码放入`try{}`块中，并使用`catch(){}`处理程序与之关联。如果`catch()`块捕获到了异常，我们将对异常情况做相应的处理。

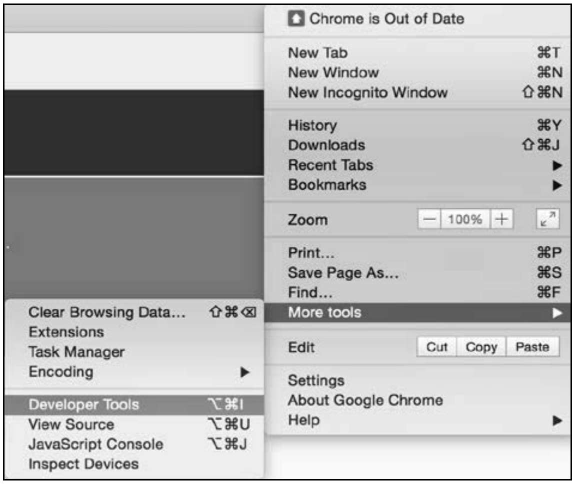
1. Console.log与断言

在控制台中显示执行状态非常有助于调试。不过，现代的开发者工具允许你设置断点，暂停执行，然后检查运行期间某个特定的值。你可以通过在控制台中记录下变量的状态，来快速地找出一些细小的问题。

了解了这些概念之后，让我们来看看如何使用Chrome的Developer Tools调试JavaScript代码。

2. Chrome DevTools

可以通过菜单| More tools | Developer Tools来启动Chrome DevTools：



Chrome DevTools会在浏览器底部开启一个面板，其中包含很多非常有用的区域：



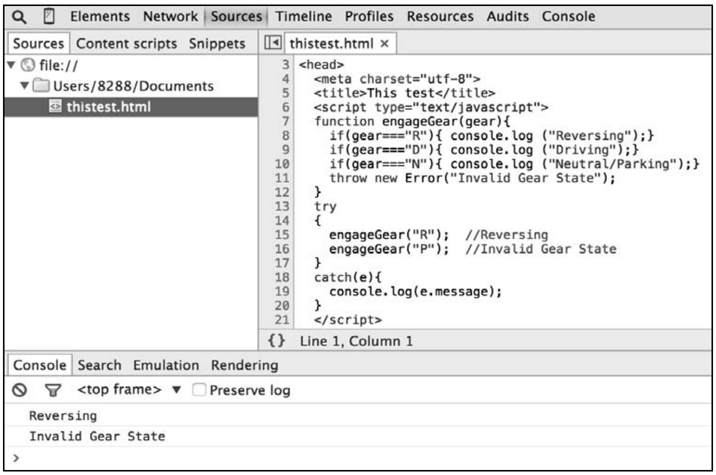
Elements面板可以帮助你检查和监视DOM树以及相关的样式表单。

Network面板有助于理解网络活动。例如，你可以实时地监视网络上下载的资源。

对于我们而言，最重要的就是Sources面板，JavaScript源代码以及调试器会在该面板中显示。下面来创建一个HTML样例：

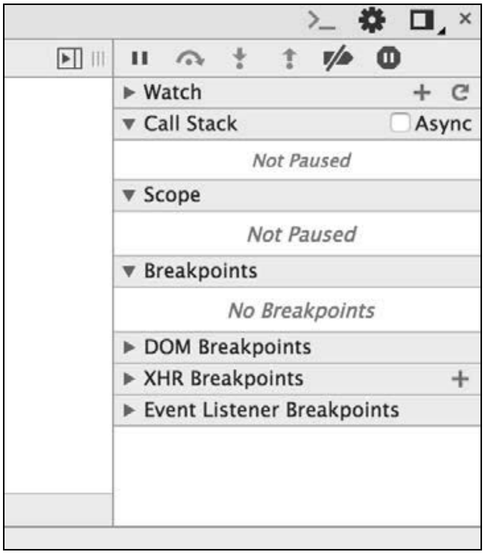
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
  function engageGear(gear){
    if(gear==="R"){ console.log ("Reversing");}
    if(gear==="D"){ console.log ("Driving");}
    if(gear==="N"){ console.log ("Neutral/Parking");}
    throw new Error("Invalid Gear State");
  }
  try
  {
    engageGear("R"); //Reversing
    engageGear("P"); //Invalid Gear State
  }
  catch(e){
    console.log(e.message);
  }
  </script>
</head>
<body>
</body>
</html>
```

保存该HTML文件，并在Google Chrome中打开。在浏览器中启动DevTools，会看到如下画面：

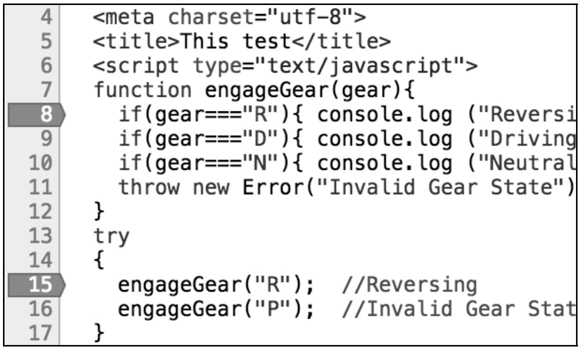


这就是Sources面板的视图。可以在其中看到HTML及嵌入的JavaScript源代码，还可以看到Console窗口。所执行的文件及其输出会显示在Console中。

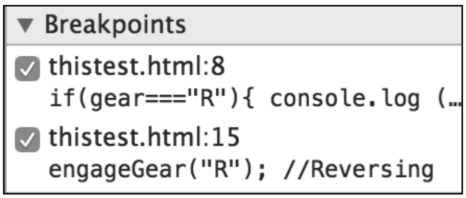
右手边就是调试器窗口：



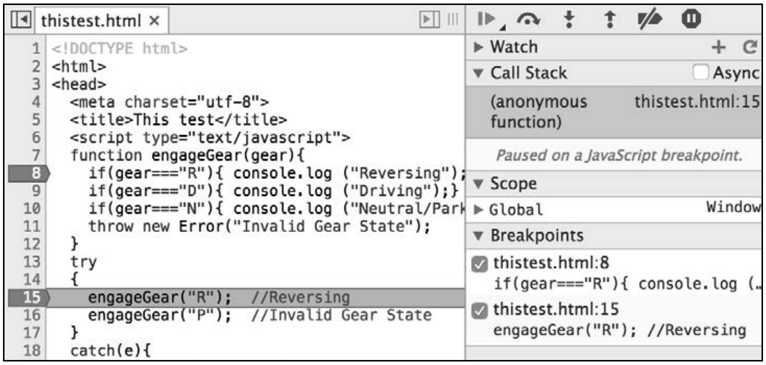
在Sources面板中，点击第8行和第15行来添加断点。断点可以在指定位置停止脚本的执行：



在调试面板中，你可以看到所有的断点：



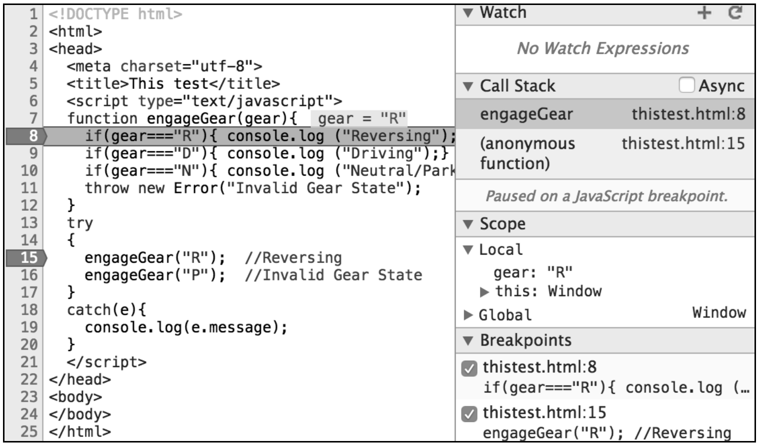
现在，当你重新运行相同的页面时，会发现执行过程在断点处停下来了。一个非常有用的技巧是在调试阶段插入代码。在调试器运行期间，可以通过添加代码来帮助你更好地理解代码的状态：



这个窗口已经万事皆备了。你可以看到执行过程暂停在第15行。在调试窗口中，能够知道触发的是哪个断点。另外还可以看到Call Stack。恢复脚本执行的方法有好几种。调试命令窗口有一系列操作：



可以通过点击按钮来恢复执行（直至下一个断点）。当执行此操作时，脚本会继续执行，直到碰到下一个断点。在这个例子中，会在第8行停住。



Call Stack窗口显示出了是如何执行到第8行代码的。Scope面板显示出了Local作用域，你可以在断点被触发时观察到该作用域中的变量。还可以对下一个函数选择步进（step into）或是步跳（step over）操作。

Chrome DevTools中还有其他一些非常有用的机制可以用于代码的调试和分析。建议你多用DevTools练练手，使它成为你日常开发流程的一部分。

6.3　小结

要想写出稳固的JavaScript代码，测试和调试是必不可少的阶段。TDD和BDD与敏捷方法论密不可分，已经被JavaScript开发者社区广泛接受。在本章中，我们了解了TDD的最佳实践以及测试框架Jasmine的用法，看到了使用Chrome DevTools进行JavaScript代码调试的各种方法。在下一章中，我们将对新奇的ES6世界、DOM操作以及跨浏览器策略一探究竟。

第 7 章

ECMAScript 6

到目前为止，我们详细学习了JavaScript编程语言。我相信你对这门语言的核心已经有了深刻的领悟。到目前为止，我们看到的都是ECMAScript 5（ES5）中的内容。ECMAScript 6（ES6）或ECMAScript 2015（ES2015）是ECMAScript标准的最新版本。这套标准一直在演进，最近一轮修改是在2015年6月完成的。ES2015成效显著，其规范已经在大多数JavaScript引擎中实现。这可是个好消息。ES6引入了大量的特性，增添了新的语法和辅助工具（helper），大大地丰富了这门语言。浏览器和JavaScript引擎支持新特性的速度，多少有点跟不上ECMAScript标准演进的步伐。摆在眼前的一个现实问题是，绝大多数程序员不得不编写旧浏览器能支持的代码。臭名昭著的Internet Explorer 6曾经是全世界使用最广泛的浏览器，要保证代码与绝大部分浏览器兼容可是一件恐怖的任务。所以，尽管你想一跃跳入ES6那些令人赞不绝口的新特性中，但是你也得考虑一个事实：ES6的一些特性可能还未被最流行的浏览器或JavaScript框架所支持。

这实在是让人痛苦，不过事情不是没有转机。Node.js采用的最新的V8引擎支持大部分ES6特性，Facebook的React对ES6的支持力度也不差。作为现今使用率最高的两款浏览器，Mozilla Firefox和Google Chrome也支持了大部分ES6特性。

为了避免此类陷阱以及不可预测性，已经提出了一些解决方案。其中最有用的就是polyfill/shim和转换编译器。

7.1 shim/polyfill

polyfill（也称为shim）[①]是一种模式，它采用旧环境所支持的兼容形式来定义新环境的行为。在GitHub上有一个叫作ES6 shim的地方（https://github.com/paulmillr/es6-shim/），那里集合了一大批ES6 shim，强烈建议你去学习一下。考虑一个ES6 shim的例子。

ECMAScript 2015（ES6）标准中的`Number.isFinite()`方法可以判断传入的值是否是一个有限数字，其等价的shim如下：

① 无论是polyfill还是shim，目前都没有统一的译法，故在本书中对这两个词保留了原文。（在由人民邮电出版社出版的《HTML5秘籍（第2版）》中将polyfill译为“腻子脚本”，个人认为是一个不错的译法。）——译者注

```
var numberIsFinite = Number.isFinite || function isFinite(value) {
  return typeof value === 'number' && globalIsFinite(value);
};
```

这个shim首先判断`Number.isFinite()`方法是否可用；如果不可用，它使用其他的实现来补位。这种技术真是太漂亮了，它填补了规范留下的空白。shim会伴随着新特性不断升级，因此，在项目中使用最新的shim可谓是一种明智的策略。

> 关于`endsWith()`的polyfill的详细描述可以在https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith找到。`String.endWith()`是ES6的一部分，不过很容易通过polyfill在ES6之前的环境中使用。

但shim并非包治百病，它无法应对语法上的变化。这时候，就可以考虑使用转换编译器了。

7.2　转换编译器

转换编译（transpiling）是一种结合了编译和转换的技术。其思想是编写与ES6兼容的代码，然后使用工具将其转换编译成有效的ES5等价代码。下面来看一款功能最完备，也是最流行的ES6转换编译器（transpiler）Bable（https://babeljs.io/）。

Bable的用法不止一种。你可以将其安装成node模块，从命令行调用；也可以作为脚本导入到Web页面中。https://babeljs.io/docs/setup/上详细地描述了Bable的安装过程。Bable也有一个不错的**读取–求值–输出–循环**（Read-Eval-Print-Loop，REPL）环境，我们将使用Bable的REPL来演示本章中的大多数例子。深入理解Bable的各种用法超出了本书的范围，不过我建议你现在就开始将Bable作为开发工作流程的一部分。

我们将在本章中介绍ES6规范最重要的部分。你应该尽可能去尝试ES6所有的特性，并将其融入开发流程中。

7.3　ES6 语法上的变化

ES6给JavaScript语法带来了巨大的变化，这些变化需要仔细研究并加以习惯。在本节中，我们将学习一些最重要的语法变化，看看如何利用Bable在代码中立刻使用这些新的语言构件。

7.3.1　块级作用域

之前讲过，JavaScript中的变量都是函数作用域。在嵌套作用域中创建的变量可用于整个函数。有些编程语言提供了默认的块级作用域，任何在代码块中（通常由`{}`分隔）声明的变量被限制（可

用于）在该块中。为了在JavaScript中实现类似的块级作用域，一个流行的方法是利用**立即调用函数表达式**（immediately-invoked function expression，IIFE）。考虑下面的例子：

```
var a = 1;
(function blockscope(){
  var a = 2;
  console.log(a); // 2
})();
console.log(a); // 1
```

有了IIFE，我们就可以为变量a创建一个块级作用域。对于在IIFE中声明的变量，其作用域被限制在函数内。这是模拟块级作用域的传统方法。ES6对块级作用域提供了直接支持，不需要再用IIFE了。在ES6中，你可以在{}所定义的语句块中放入任何语句。在声明具有块级作用域的变量时，不再使用var，而是改用let。上面的例子可以使用ES6块级作用域改写成如下形式：

```
"use strict";
var a = 1;
{
  let a = 2;
  console.log( a ); // 2
}
console.log( a ); // 1
```

单独使用{}在JavaScript中似乎并不多见，但是这种创建块级作用域的习惯用法在很多语言中很常见。块级作用域也用于其他语言构件中，如if{ }或for{ }。

在以这种方式使用块级作用域时，一般更倾向于将变量声明置于语句块的顶部。使用var声明变量和使用let声明变量的一个区别就在于，前者是函数作用域，后者是块级作用域，而且是在块中所出现的位置上初始化。因此对于使用let声明的变量，你无法在其声明之前进行访问；而对于使用var声明的变量，访问顺序并不重要：

```
function fooey() {
  console.log(foo); // ReferenceError
  let foo = 5000;
}
```

let在循环中有一种特定的用法。如果在for循环中使用由var声明的变量，该变量是在全局作用域或父作用域中创建的。我们可以通过let在for循环中声明一个块级作用域变量。考虑下面的例子：

```
for (let i = 0; i<5; i++) {
  console.log(i);
}
console.log(i); // i没有定义
```

因为i是使用let创建的，故其作用域被限制在for循环中，该变量在作用域之外是不可用的。

ES6中块级作用域还可以用来创建常量。使用关键字const，可以在块级作用域中创建常量。

一旦设置好常量的值，就再也不能更改了：

```
if(true){
  const a=1;
  console.log(a);
  a=100; // a是只读的，对其赋值会引发TypeError
}
```

常量在声明的同时必须初始化。块级作用域规则也适用于函数。在块中声明的函数，只能在块级作用域范围内使用。

7.3.2　默认参数

默认（defaulting）是一种很常见的做法。你总是会给传入函数的参数设置默认值或是对变量初始化。下面的代码你也许并不陌生：

```
function sum(a,b){
  a = a || 0;
  b = b || 0;
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9));   //9
```

这里，如果在调用函数时，没有为参数a和b提供对应的值，我们使用||（OR操作符）将两者的默认值设置为0。在ES6中，有一种为函数参数设置默认值的标准做法。上面的例子可以改写成：

```
function sum(a=0, b=0){
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9));   //9
```

可以将任何有效的表达式或函数调用作为默认参数列表的一部分。

7.3.3　spread 与 rest

ES6中增添了一个新操作符...。根据用法的不同，可以叫作spread或rest。来看一个小例子：

```
function print(a, b){
  console.log(a,b);
}
print(...[1,2]); //1,2
```

如果将...放在数组（或是其他可迭代的内容）之前，该操作符会将数组中的元素**分散**（spread）到函数的各个参数中。参数a和b分别从数组中得到了两个值。在分散数组元素时，会

忽略掉多余的参数：

```
print(...[1,2,3 ]); //1,2
```

这仍然会打印出1和2，因为可用的函数参数只有两个。这种用法也可以用于其他场景，例如数组赋值：

```
var a = [1,2];
var b = [ 0, ...a, 3 ];
console.log( b ); //[0,1,2,3]
```

操作符...的另一种用法和我们刚刚看到的截然相反。这次不再是将值分散，而是将值汇集（gather）到一处：

```
function print (a,...b){
  console.log(a,b);
}
console.log(print(1,2,3,4,5,6,7)); //1 [2,3,4,5,6,7]
```

在这个例子中，变量b接收了剩余的（rest）所有值。变量a接收到了第一个值1，其余的值以数组的形式传给了变量b。

7.3.4　解构

如果你用过像Erlang这样的函数式语言，对模式匹配的概念应该不陌生。JavaScript中的解构（destructuring）与此非常类似。解构允许你使用模式匹配将值与变量绑定在一起。考虑下面的例子：

```
var [start, end] = [0,5];
for (let i=start; i<end; i++){
  console.log(i);
}
//打印出0,1,2,3,4
```

利用数组解构给两个变量赋值：

```
var [start, end] = [0,5];
```

在上面的例子中，我们想要的模式就是第一个值赋给第一个变量（start），第二个值赋给第二个变量（end）。下面的代码片段展示了数组解构的方法：`

```
function fn() {
  return [1,2,3];
}
var [a,b,c]=fn();
console.log(a,b,c); //1 2 3
// 我们可以跳过其中一个
var [d,,f]=fn();
console.log(d,f);   //1 3
// 没有用到余下的值
```

```
var [e,] = fn();
console.log(e);   //1
```

来看看对象解构是怎么回事。假设有一个能够返回对象的函数`f`：

```
function f() {
  return {
    a: 'a',
    b: 'b',
    c: 'c'
  };
}
```

在解构函数返回的对象时，可以使用和先前类似的语法，不同之处在于使用的是`{}`，而非`[]`：

```
var { a: a, b: b, c: c } = f();
console.log(a,b,c); //a b c
```

和数组类似，我们使用模式匹配来将函数返回的值赋给对应的变量。如果使用的变量名和被匹配的属性名一样，那么还有一种更简短的写法。下面的例子就是这样的，没有任何问题：

```
var { a,b,c } = f();
```

不过在大部分时间里，使用的变量名与函数返回的属性名都是不一样的。重要的是要记得语法是source: destination，而不是惯常的destination: source。仔细观察下面的例子：

```
// target: source - 错误写法
var { x: a, x: b, x: c } = f();
console.log(x,y,z); // x、y、z都未定义
// source: target - 正确写法
var { a: x, b: y, c: z } = f();
console.log(x,y,z); // a b c
```

这和给变量赋值所采用的target = source正好相反，因此需要花点时间来适应。

7.3.5　对象字面量

7

在JavaScript中，对象字面量无处不在。你会觉得没什么改进的空间了，但ES6依然想对此作出改进。ES6引入了一些简写形式，于是便有了另一种更紧凑的对象字面量语法：

```
var firstname = "Albert", lastname = "Einstein",
  person = {
    firstname: firstname,
    lastname: lastname
  };
```

如果你希望赋值的属性名和变量名一样，可以使用ES6中这种简洁的属性写法：

```
var firstname = "Albert", lastname = "Einstein",
  person = {
```

```
    firstname,
    lastname
  };
```

与此类似，如果要将函数分配给属性：

```
var person = {
  getName: function(){
    // ..
  },
  getAge: function(){
    //..
  }
}
```

不用像上面那样，只需要：

```
var person = {
  getName(){
    // ..
  },
  getAge(){
    //..
  }
}
```

7.3.6　模板字面量

我相信你肯定这么干过：

```
function SuperLogger(level, clazz, msg){
  console.log(level+": Exception happened in class:"+clazz+" - Exception:"+ msg);
}
```

这是一种通过替换变量值来形成字符串字面量的极为常见的做法。ES6提供了一种使用反引号（`）作为分隔符的新形式的字符串字面量。你可以利用变量插值在模板字符串字面量中放置占位符（placeholder），这些占位符会被解析及求值。

刚才的例子可以重写为：

```
function SuperLogger(level, clazz, msg){
  console.log(`${level} : Exception happened in class: ${clazz} - Exception: {$msg}`);
}
```

我们把字符串字面量放在了``里面。在该字面量内，所有`${..}`形式的表达式会被立刻解析，这种解析过程叫作插值（interpolation）。在解析的时候，变量值将替换`${}`中的占位符。由此得到的最终结果是，一个用实际的变量值将占位符替换掉之后的普通字符串。

使用字符串插值，也可以将一个字符串拆分成多行，如下所示（很像Python的写法）：

```
var quote =
`Good night, good night!
Parting is such sweet sorrow,
that I shall say good night
till it be morrow.`;
console.log( quote );
```

可以将函数调用或有效的JavaScript表达式作为字符串插值的一部分：

```
function sum(a,b){
  console.log(`The sum seems to be ${a + b}`);
}
sum(1,2); //The sum seems to be 3
```

模板字符串的最后一种用法叫作**标记字符串模板**（tagged template string）。其思想是使用函数修改模板字符串。考虑下面的例子：

```
function emmy(key, ...values){
  console.log(key);
  console.log(values);
}
let category="Best Movie";
let movie="Adventures in ES6";
emmy`And the award for ${category} goes to ${movie}`;

//["And the award for "," goes to ",""]
//["Best Movie","Adventures in ES6"]
```

最陌生的部分就是使用模板字面量调用函数emmy，这和传统的函数调用语法不同。我们并没有编写emmy()，只是使用函数**标记**（tagging）了这个字面量。当函数被调用时，第一个参数是由所有普通字符串组成的数组（数组元素为插值表达式之间的子串），第二个参数是由所有求值后的插值表达式组成的数组。

这就意味着标记函数能够改变最终的模板字符串：

```
function priceFilter(s, ...v){
  // 提高折扣
  return s[0]+ (v[0] + 5);
}
let default_discount = 20;
let greeting = priceFilter `Your purchase has a discount of ${default_discount}
percent`;
console.log(greeting);   //Your purchase has a discount of 25
```

如你所见，我们在标记函数中修改了折扣值并将修改后的结果返回。

7.3.7　Map 与 Set

ES6引入了四个新的数据结构：Map、WeakMap、Set和WeakSet。我们之前讨论过，对象是

在JavaScript中创建键值对的惯常做法，对象的不足之处在于无法使用非字符串值作为键。下面的代码片段演示了如何在ES6中创建Map：

```
let m = new Map();
let s = { 'seq' : 101 };

m.set('1','Albert');
m.set('MAX', 99);
m.set(s,'Einstein');

console.log(m.has('1')); //true
console.log(m.get(s));   //Einstein
console.log(m.size);     //3
m.delete(s);
m.clear();
```

可以在声明的同时初始化：

```
let m = new Map([
  [ 1, 'Albert' ],
  [ 2, 'Douglas' ],
  [ 3, 'Clive' ],
]);
```

如果想迭代Map中的条目，可以使用函数`entries()`，它会返回一个迭代器。你可以使用函数`key()`迭代所有的键，使用函数`values()`迭代所有的值：

```
let m2 = new Map([
  [ 1, 'Albert' ],
  [ 2, 'Douglas' ],
  [ 3, 'Clive' ],
]);
for (let a of m2.entries()){
  console.log(a);
}
//[1,"Albert"] [2,"Douglas"][3,"Clive"]
for (let a of m2.keys()){
  console.log(a);
} //1 2 3
for (let a of m2.values()){
  console.log(a);
}
//Albert Douglas Clive
```

JavaScript Map的另一种形式是WeakMap。一个WeakMap无法阻止它的键被垃圾回收机制处理。WeakMap的键必须是对象，值的类型没有限制。尽管行为表现和普通的Map无异，但你不能对其进行迭代，也无法清除它。这些限制的背后有一定的原因。因为Map的状态未必一直是静态（键可能会被垃圾回收机制处理），故无法保证正确的迭代操作。

要用到WeakMap的情形并不多。Map的大部分用法都可以使用普通的Map来实现。

Map允许存储各种类型的值，而Set是不重复值的集合。Set的有些方法和Map类似；但set()方法变成了add()，也没有get()方法。没有get()方法的原因在于Set中的值都是唯一的，所以如果需要的话，只检查Set是否包含某个值就行了。考虑下面的例子：

```
let x = {'first': 'Albert'};
let s = new Set([1,2,'Sunday',x]);
//console.log(s.has(x)); //true
s.add(300);
//console.log(s);   //[1,2,"Sunday",{"first":"Albert"},300]

for (let a of s.entries()){
  console.log(a);
}
//[1,1]
//[2,2]
//["Sunday","Sunday"]
//[{"first":"Albert"},{"first":"Albert"}]
//[300,300]
for (let a of s.keys()){
  console.log(a);
}
//1
//2
//Sunday
//{"first":"Albert"}
//300
for (let a of s.values()){
  console.log(a);
}
//1
//2
//Sunday
//{"first":"Albert"}
//300
```

迭代器keys()和values()都能够返回Set中不重复值的列表。entries()返回一个条目数组列表，数组中的项都是不重复的值。Set()默认的迭代器是values()。

7

7.3.8　Symbol

ES6引入了一种新的数据类型Symbol，它是唯一且不可变的值。Symbol通常用作对象属性的标识符，可以将其看作唯一的ID。你可以使用工厂方法Symbol()创建Symbol。但是要记住，Symbol()并不是构造函数，不需要使用new操作符：

```
let s = Symbol();
console.log(typeof s); //symbol
```

与字符串不同，Symbol可以确保唯一性，因此有助于避免名称冲突。利用Symbol，我们就有了一个可扩展的机制。ES6包含很多内建的Symbol值，这些值能够向用户暴露出JavaScript对象

的各种元行为（meta behavior）。

7.3.9　迭代器

迭代器在其他编程语言中已经存在一段时间了，它们为处理数据集合提供了非常方便的方法。ES6也是出于同样的目的引入了迭代器。ES6的迭代器是具有特定接口的对象。迭代器的`next()`方法可以返回一个对象。该对象有两个属性：value（下一个值）和done（指示是否已经抵达最后一个结果）。ES6还定义了一个`Iterable`接口，该接口描述了能够产生迭代器的对象。来看一个可迭代的数组，它所产生的迭代器可以用来处理数组元素：

```
var a = [1,2];
var i = a[Symbol.iterator]();
console.log(i.next());        // { value: 1, done: false }
console.log(i.next());        // { value: 2, done: false }
console.log(i.next());        // { value: undefined, done: true }
```

如你所见，我们通过`Symbol.iterator()`来访问数组的迭代器，并调用其`next()`方法来获取每一个后续元素。该方法会返回`value`和`done`。当你在数组最后一个元素之外调用`next()`方法时，会得到一个undefined值以及`done: true`，表明整个数组已经迭代完毕。

7.3.10　`for..of` 循环

ES6加入了一个新的迭代机制：`for..of`循环，它可以遍历由迭代器生成的一组值。

使用`for..of`所遍历的值是可迭代的。

来比较一下`for..of`和`for..in`：

```
var list = ['Sunday','Monday','Tuesday'];
for (let i in list){
  console.log(i);   //0 1 2
}
for (let i of list){
  console.log(i);   //Sunday Monday Tuesday
}
```

可以看到，使用`for..in`循环的话，可以遍历数组`list`的索引，而`for..of`循环可以遍历数组`list`的值。

7.3.11　箭头函数

ECMAScript 6新增特性中最值得注意的一个部分就是箭头函数（arrow function）。正如其名，箭头函数使用**箭头**（`=>`）作为函数定义语法的组成部分。先来看看箭头函数的样子：

```
// 传统函数
function multiply(a,b) {
  return a*b;
}
// 箭头函数
var multiply = (a,b) => a*b;
console.log(multiply(1,2)); //2
```

箭头函数的定义由参数列表［零个或多个参数，如果不止一个参数，要在参数外加上 (..)］、=>符号以及随后的函数体所组成。

如果函数体中不止一个表达式，需要在函数体外加上{ .. }。如果只有一个表达式的话，可以忽略周围的{ .. }，在表达式之前会有一个隐含的换行。箭头函数由多种不同的写法，下面是最常用的几种：

```
// 单个参数，单条语句
// arg => expression;
var f1 = x => console.log("Just X");
f1(); //Just X

// 多个参数，单条语句
// (arg1 [, arg2]) => expression;
var f2 = (x,y) => x*y;
console.log(f2(2,2)); //4

// 单个参数，多条语句
// arg => {
//     statements;
// }
var f3 = x => {
  if(x>5){
    console.log(x);
  }
  else {
    console.log(x+5);
  }
}
f3(6); //6

// 多个参数，多条语句
// ([arg] [, arg]) => {
// statements
// }
var f4 = (x,y) => {
  if(x!=0 && y!=0){
    return x*y;
  }
}
console.log(f4(2,2));//4

// 没有参数，单条语句
//() => expression;
```

7

```
var f5 = () => 2*2;
console.log(f5()); //4

//IIFE
console.log(( x => x * 3 )( 3 )); // 9
```

重要的是要记住普通函数参数的所有特点都可以用于箭头函数，这包括默认值、解构以及rest参数。

箭头函数提供了一种简洁的语法，让你的代码很有**函数式编程**的范儿。它之所以受欢迎的原因在于从代码中摒弃了funcion、return以及`{ .. }`，更易于编写出紧凑的函数。然而，箭头函数通过`this`感知（this-aware coding）从根本上解决了一个特别的常见痛点。在普通的ES5函数中，每一个新函数都定义了自己的`this`值（在构造函数中是新对象，在严格模式下的函数调用中是`undefined`，如果是以**对象方法**形式调用的函数，则是上下文对象……）。JavaScript函数总是有自己的`this`值，这使得我们无法在回调函数中访问外围方法的`this`。要想搞明白这个问题，考虑下面的例子：

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){     // --> 1
  'use strict';
  return s.map(function (a){               // --> 2
    return this.str + a;                   // --> 3
  });
};

var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//Cannot read property 'str' of undefined
```

在标记为3的一行中，我们试图获得`this.str`，但是匿名函数也有自己的`this`，正好将行`1`方法中的`this`遮盖（shadow）了。要想在ES5中修正这个问题，需要将`this`赋给一个变量，然后用这个变量来代替：

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){
  'use strict';
  var that = this;                         // --> 1
  return s.map(function (a){               // --> 2
    return that.str + a;                   // --> 3
  });
};

var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//["HelloWorld]
```

标记为1的行中，我们将this赋给了一个变量that，并将其用于匿名函数中，这样就能够引用到所需要的上下文的this值了。

ES6的箭头函数有一个词法范围的this（lexical this），意味着箭头函数能够获得外围上下文（enclosing context）的this值。我们可以将上述函数改写成等价的箭头函数形式：

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){
  return s.map((a)=> {
    return this.str + a;
  });
};
var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//["HelloWorld]
```

7.4　小结

在本章中，我们讨论了ES6中的一些重要的新特性。这是一组令人激动的新语言特性和范式，利用polyfill和转换编译器，你立刻就可以上手把玩。JavaScript是一门不断发展的语言，理解其未来发展很重要。ES6所增添的特性使得JavaScript变得愈发有趣和成熟。在下一章中，我们将使用jQuery和JavaScrip深入探索浏览器的**文档对象模型**（Document Object Model，DOM）操作及事件。

第 8 章 DOM操作与事件

JavaScript存在的最重要原因就是Web。JavaScript是一门属于Web的语言，浏览器是JavaScript表演的舞台。如果少了JavaScript，我们看到的就只能是静态的Web页面。在本章中，我们将深入挖掘JavaScript与浏览器之间的关联，理解其与Web页面各组成部分之间的交互方式，学习**文档对象模型**（Document Object Model，DOM）和JavaScript事件模型。

8.1 DOM

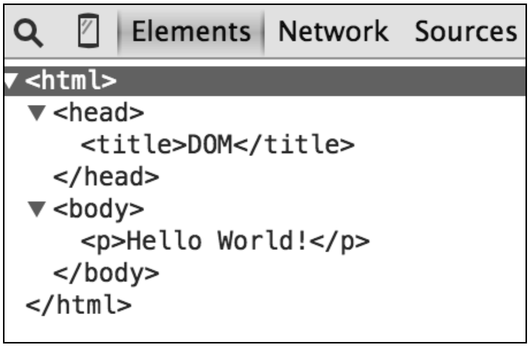
在本章中，我们将学习JavaScript与浏览器和HTML相关的各个方面。我相信你一定知道，HTML是用于编写Web页面的一种标记语言。不同的标记语言有不同的用途。流行的标记语言包括**可扩展标记语言**（eXtensible Markup Language，XML）和**标准通用标记语言**（Standard Generalized Markup Language，SGML）。除了这些通用型标记语言，还有一些具有特定用途（比如文本处理和图像元信息）的专用标记语言。**超文本标记语言**（HyperText Markup Language，HTML）是用于定义Web页面语义的标准标记语言。Web页面实际上就是一份文档，DOM提供了这份文档的描述信息。除此之外，它还有一套用于存储和操作文档的方法。DOM是HTML的编程接口，允许使用JavaScript等脚本语言对其进行结构化的操作。文档的结构化描述也是由DOM提供的，该结构由节点和对象组成。节点拥有属性和方法，可用于操作节点本身。不过DOM仅仅是一种描述，并非编程构件。它作为一种模型，供能够处理DOM的语言使用，如JavaScript。

8.1.1 访问 DOM 元素

大多数情况下，你需要根据某些业务逻辑访问DOM元素来检查或处理相应的值。我们将详细讨论这种用法。创建一个HTML文件样例，内容如下：

```
<html>
<head>
  <title>DOM</title>
</head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

将文件保存为sample_dom.html。在Google Chrome浏览器中打开该文件，会看到页面上显示出文本Hello World。现在，通过选项中的 | More Tools | Developer Tools（具体步骤可能会随操作系统或浏览器的版本而不同）打开Google Chrome Developer Tools。在Developer Tools窗口中，会看到如下DOM结构：



接下来，可以向页面中插入一些JavaScript代码。我们打算在页面载入后调用一个JavaScript函数。这可以通过在`window.onload`上调用函数来实现。把脚本放入`<script>`标签中，然后将其放在`<head>`标签之后。页面如下所示：

```
<html>
  <head>
    <title>DOM</title>
    <script>
      // 当文档载入完成时执行该函数
      window.onload = function() {
        var doc = document.documentElement;
        var body = doc.body;
        var _head = doc.firstChild;
        var _body = doc.lastChild;
        var _head_ = doc.childNodes[0];
        var title = _head.firstChild;
        alert(_head.parentNode === doc); //true
      }
    </script>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```


匿名函数会在浏览器载入页面后执行。在这个函数中，我们以编程的方式获取了DOM节点。整个HTML文档可以通过`document.documentElement`访问。我们将文档保存到一个变量中。在访问文档时，利用文档的一些辅助属性来遍历这些节点。我们使用`doc.body`访问`<body>`元素，利用`childNodes`数组来遍历某个元素的子节点，某个节点的第一个和最后一个子节点可以使用`firstChild`和`lastChild`属性访问。

> 不推荐在<head>标签中使用阻塞渲染的JavaScript，这会严重拖慢页面的渲染速度。现代浏览器都支持async和defer属性，用于指示浏览器在下载脚本的同时继续渲染页面。可以在<head>标签中使用这些属性，无需再担心影响性能。更多的相关信息可以在这里找到：http://stackoverflow.com/questions/436411/where-is-the-best-place-to-put-script-tags-in-html-markup。

8.1.2　访问特定的节点

核心DOM定义了getElementsByTagName()方法，它能够返回一个NodeList，其中包含了tagName属性与指定值相同的所有元素对象。下面这行代码会返回由文档中所有<p/>元素组成的列表：

```
var paragraphs = document.getElementsByTagName('p');
```

HTML DOM定义了getElementsByName()方法，能够检索到所有name特性（attribute）[1]与指定值相同的元素。考虑下面的代码片段：

```
<html>
  <head>
    <title>DOM</title>
    <script>
      showFeelings = function() {
        var feelings = document.getElementsByName("feeling");
        alert(feelings[0].getAttribute("value"));
        alert(feelings[1].getAttribute("value"));
      }
    </script>
  </head>
  <body>
    <p>Hello World!</p>
    <form method="post" action="/post">
      <fieldset>
        <p>How are you feeling today?</p>
        <input type="radio" name="feeling" value="Happy" /> Happy<br />
        <input type="radio" name="feeling" value="Sad" />Sad<br />
      </fieldset>
      <input type="button" value="Submit" onClick="showFeelings()"/>
    </form>
  </body>
</html>
```

在本例中，我们创建了一组name特性为feeling的单选按钮。在函数showFeelings()中，获取到所有name特性为feeling的元素并对其进行迭代。

① 为了便于区分，本书中将property译为“属性”，将attribute译为“特性”。——译者注

HTML DOM还定义了另一个叫作getElementById()的方法。在访问特定元素的时候，这个方法非常有用。它根据元素的id进行查找。id特性对于每个元素都是唯一的，因此这种查找过程速度非常快，优于getElementsByName()。不过要留意的是浏览器并不保证id特性的唯一性。在下面的例子中，我们利用ID来访问特定的元素。与标签或name特性相反，元素的ID是唯一的：

```
<html>
  <head>
    <title>DOM</title>
    <script>
      window.onload= function() {
        var greeting = document.getElementById("greeting");
        alert(greeting.innerHTML); //显示提示框"Hello World"
      }
    </script>
  </head>
  <body>
    <p id="greeting">Hello World!</p>
    <p id="identify">Earthlings</p>
  </body>
</html>
```

到目前为止，我们讨论的都是JavaScript中DOM遍历的基础知识。随着DOM的结构及其操作变得愈发复杂，这些用来遍历和访问的函数所发挥的作用就有限了。有了这些基础，就可以介绍一个叫作jQuery的库，它可用于DOM遍历（还有其他用途），功能极其强大。

jQuery是一个轻量级的库，旨在简化常见的浏览器操作。这些常见操作包括DOM遍历及操作、事件处理、动画和Ajax，如果使用纯JavaScript来实现的话，那可是一个令人乏味的过程。jQuery为你提供了易用和更简短的辅助工具机制，帮助你简单快速地掌握这些常见操作。尽管功能丰富，考虑到本章的侧重点，我们主要关注DOM操作和事件。

有两种方法可以将jQuery添加到HTML中：通过直接从**内容分发网络**（content delivery network，CDN）载入脚本，或是手动下载库文件并将其放入<script>标签。下面的例子展示了如何从Google的CDN中下载jQuery：

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
        jquery.min.js"></script>
  </head>
  <body>
  </body>
</html>
```

使用CDN下载的优势在于，Google的CDN能够自动为你找到距离最近的下载服务器，保持jQuery库处于最新的稳定状态。如果你希望下载并手动将jQuery安装在自己的网站中，可以加入如下内容：

8

```
<script src="./lib/jquery.js"></script>
```

在本例中，jQuery库被手动下载到了lib目录。在HTML页面中设置好jQuery之后，就让我们一探那些可用来操作DOM元素的方法吧。考虑下面的例子：

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
        jquery.min.js"></script>
    <script>
    $(document).ready(function() {
        $('#greeting').html('Hello World Martian');
    });
  </script>
  </head>
  <body>
    <p id="greeting">Hello World Earthling ! </p>
  </body>
</html>
```

向HTML页面中加入jQuery之后，我们编写了一个定制的JavaScript脚本，用于选择ID为`greeting`的元素并改变对应的值。`$()`中样子有些陌生的代码就是用来完成这项任务的jQuery。如果阅读jQuery的源代码（你应该去读读，简直太棒了），会发现最后一行：

```
// 将jQuery暴露给全局对象
window.jQuery = window.$ = jQuery;
```

`$`只是一个函数，它是名为jQuery的函数的别名。`$`就是一个语法糖，为了使代码紧凑而已。实际上可以交替使用`$`和`jQuery`。例如，`$('#greeting').html('Hello World Martian');`和`jQuery('#greeting').html('Hello World Martian');`的效果是一样的。

只有在页面完全载入之后才能使用jQuery。因为jQuery需要知道DOM结构的所有节点，所以完整的DOM必须处于内存中。为了确保页面完全载入并进入可操作状态，我们可以使用`$(document).ready()`函数。下面的IIFE只会在整个页面已经**准备完毕**（ready）之后才执行：

```
$(document).ready(function() {
  $('#greeting').html('Hello World Martian');
});
```

这段代码展示了如何将一个函数与jQuery的`.ready()`函数关联在一起。这个函数会在文档准备好之后执行。我们使用`$(document)`为页面文档创建一个对应的jQuery对象，然后在jQuery对象上调用`.ready()`，并将其传入待执行的函数：

这种操作在使用jQuery时极为常见，以至于它拥有相应的简写方式。可以用简短的`$()`调用来代替整个`ready()`：

```
$(function() {
  $('#greeting').html('Hello World Martian');
});
```

`$()`是jQuery中最重要的函数。它通常接受CSS选择器作为唯一的参数，返回一个新的jQuery对象，该对象指向选择器所匹配到的对应页面元素。三种主要的选择器是标签名、ID和类。它们既可以单独使用，也可以结合使用。下面一些简单的例子演示了这三种选择器在代码中的使用方式：

选择器	CSS选择器	jQuery选择器	选择器的选择结果
标签	`p{}`	`$('p')`	选中文档中所有的p标签
ID	`#div_1`	`$('#div_1')`	选中文档中ID为`div_1`的单个元素。用于表示ID匹配的符号是#
类	`.bold_fonts`	`$('.bold_fonts')`	选中文档中CSS类为`bold_fonts`的所有元素。用于表示类匹配的符号是.

CSS选择器是jQuery的运行基础。

> 因为CSS选择器并不在本书的讨论范围中，所以建议你到http://www.w3.org/TR/CSS2/selector.html学习相关的概念。

我们假设你熟悉HTML标签及语法。下面的例子涵盖了jQuery选择器的基本使用方法：

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
    <script>
      $(function() {
        $('h1').html(function(index, oldHTML){
          return oldHTML + "Finally?";
        });
        $('h1').addClass('highlight-blue');
        $('#header > h1 ').css('background-color', 'cyan');
        $('ul li:not(.highlight-blue)').addClass('highlight-green');
        $('tr:nth-child(odd)').addClass('zebra');
      });
    </script>
    <style>
      .highlight-blue {
        color: blue;
      }
      .highlight-green{
        color: green;
      }
      .zebra{
        background-color: #666666;
        color: white;
      }
    </style>
  </head>
  <body>
    <div id=header>
```

```
        <h1>Are we there yet ? </h1>
        <span class="highlight">
          <p>Journey to Mars</p>
          <ul>
            <li>First</li>
            <li>Second</li>
            <li class="highlight-blue">Third</li>
          </ul>
        </span>
        <table>
          <tr><th>Id</th><th>First name</th><th>Last Name</th></tr>
          <tr><td>1</td><td>Albert</td><td>Einstein</td></tr>
          <tr><td>2</td><td>Issac</td><td>Newton</td></tr>
          <tr><td>3</td><td>Enrico</td><td>Fermi</td></tr>
          <tr><td>4</td><td>Richard</td><td>Feynman</td></tr>
        </table>
      </div>
    </body>
  </html>
```

在本例中，我们使用选择器选中了HTML页面中多个DOM元素。我们有一个内容为`Are we there yet?`的H1标题；当页面载入时，jQuery脚本会访问所有的H1标题并在其后追加文本`Finally?`：

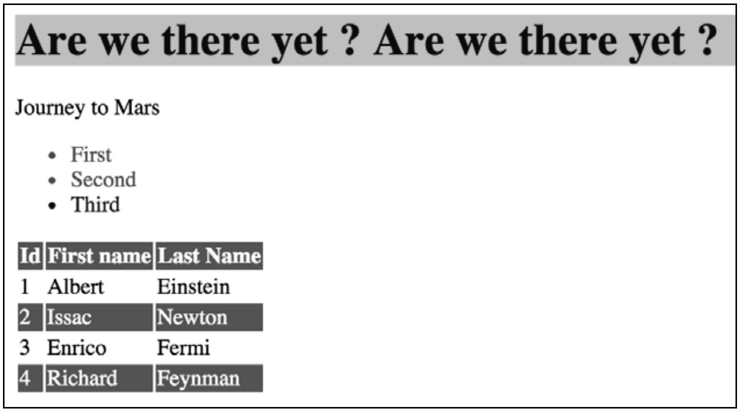
```
$('h1').html(function(index, oldHTML){
  return oldHTML + "Finally ?";
});
```

`$.html()`函数会为目标元素设置HTML——在本例中就是H1标题。另外，我们还选中了所有的H1标题并为其应用了CSS样式类`highlight-blue`。`$('h1').addClass('highlight-blue')`语句选中所有的H1标题，使用`$.addclass(<CSS class>)`方法为选中的所有元素应用CSS类。

我们使用子结合符（child combinator，`>`）配合`$.css()`函数来定制CSS样式。`$()`函数中的选择器实际上是说："所找到标题（`h1`）需要有一个ID为header（`#header`）的子元素（`>`）。"对于每一个这样的元素，我们都应用一个定制的CSS样式。接下来的这种用法值得注意。考虑下面这行代码：

```
$('ul li:not(.highlight-blue)').addClass('highlight-green');
```

我们选择了所有不包含`highlight-blue`类的列表元素（`li`），为其应用`highlight-green`类。最后一行——`$('tr:nthchild(odd)').addClass('zebra')`的意思是：为所有是奇数的表格行（`tr`）应用`zebra`类。选择器nth-child是一个由jQuery所提供的定制选择器。最终的输出结果类似于下图（尽管展示了多种jQuery选择器类型，但显然jQuery并不能代替差劲的设计品味）：



一旦做好了选择，接下来就有两类方法可以在被选中的元素上调用，这些方法就是**接收器**（getter）与**设置器**（setter）。接收器可以从所选内容中接收信息，设置器可以通过某种方式改变所选内容。

接收器通常只处理所选内容中的第一个元素，而设置器则处理所选内容中所有的元素。设置器利用隐式迭代（implicit iteration）自动遍历选中的所有元素。

例如，我们想对页面中所有的列表项应用一个CSS类。当我们在选择器上调用`addClass`时，它会自动应用到所选中的所有元素上。这就是隐式迭代的做法：

```
$( 'li' ).addClass( highlighted' );
```

但有时候你并不想通过隐式迭代遍历所有的元素，只是希望有选择地修改少数几个元素。可以使用`.each()`方法对元素进行显式迭代。在下面的代码中，我们利用元素的`index`属性，选择处理部分元素：

```
$( 'li' ).each(function( index, element ) {
  if(index % 2 == 0)
    $(elem).prepend( '<b>' + STATUS + '</b>' );
});
```

8.2　链式方法

jQuery的链式方法能够让你在不需要临时存储中间结果的情况下，在选择内容上连续调用一系列方法。这种操作可行的原因在于，每个调用的设置器方法都会返回它处理过的选择内容。这是一种非常强大的特性，很多专业库中都用到过。考虑下面的例子：


```
$( '#button_submit' )
  .click(function() {
    $( this ).addClass( 'submit_clicked' );
  })
```

```
  .find( '#notification' )
    .attr( 'title', 'Message Sent' );x
```

在这段代码中，我们在选择器上链式调用了click()、find()和attr()方法。click()方法先执行，结束之后再由find()方法找出ID为notification的元素，并将其title属性修改为字符串。

8.3　遍历与操作

我们讨论了使用jQuery选择元素的各种方法，接下来学习使用jQuery对选择内容进行DOM遍历及操作的方法。如果用原生的DOM操作完成这些任务的话，那可是相当乏味的过程。jQuery使得这一切变得既直接又优雅。

在深入了解这些方法之前，我们先来熟悉一点今后要用到的HTML术语。考虑下面的HTML：

```
<ul> <-This is the parent of both 'li' and ancestor of everything in
  <li> <-The first (li) is a child of the (ul)
    <span> <-this is the descendent of the 'ul'
      <i>Hello</i>
    </span>
  </li>
  <li>World</li> <-both 'li' are siblings
</ul>
```

利用jQuery的遍历方法，我们可以选中第一个元素，然后遍历与其相关的DOM元素。在遍历DOM的过程中，我们可以修改最初选择的内容，要么用新内容替换，要么修改原有选择。

例如，你可以过滤已有的选择，使其中只包括符合某些条件的元素。考虑下面的例子：

```
var list = $( 'li' ); // 选择所有的li元素
// 过滤出含有highlight类的li元素
var highlighted = list.filter( '.highlight );
// 过滤出不含有highlight类的li元素
var not_highlighted = list.not( '.highlight );
```

可以使用jQuery添加或删除元素的类。如果想切换元素的类，可以使用toggleClass()方法：

```
$( '#usename' ).addClass( 'hidden' );
$( '#usename' ).removeClass( 'hidden' );
$( '#usename' ).toggleClass( 'hidden' );
```

大多数情况下，你可能需要频繁地修改元素的值。可以利用val()方法来实现。例如，下面这行代码改变了所有类型为text的表单元素的值：

```
$( 'input[type="text"]' ).val( 'Enter usename:' );
```

要修改元素特性，可以使用attr()方法：

```
$('a').attr( 'title', 'Click' );
```

在处理DOM操作时，jQuery的能力深不可测。考虑到本书篇幅的限制，我们无法详细地讨论所有的用法。

8.4 处理浏览器事件

在进行浏览器相关的开发时，必须处理用户交互及其关联的事件，比如文本框中的输入、页面滚动、鼠标点击等。当用户在页面上执行了某些操作，就会发生事件。有些事件并不是由用户交互行为触发的，例如，`load`事件就不需要用户输入。

在处理浏览器中的鼠标或键盘事件时，你无法预测这些事件将在什么时候以什么顺序发生，只能不停地查看是否有键盘点击或鼠标移动出现。这就像是在后台运行了一个永不停止的循环，负责侦听是否有键盘或鼠标事件发生。在传统编程中，这叫作轮询（polling）。轮询的形式有很多种，可以使用队列来优化等待线程；但轮询仍然算不上是一种好的思路。

浏览器提供了另一种比轮询更好的方法，这种方法能够以可编程的方式应对事件的发生。这类用于事件处理的钩子（hook）通常叫作侦听器（listener）。你可以注册一个应对特定事件的侦听器，在事件被触发时执行与之关联的回调函数。考虑下面的例子：

```
<script>
  addEventListener("click", function() {
    ...
  });
</script>
```

`addEventListener`函数将第二个参数注册成回调函数，它会在由第一个参数指定的事件被触发时执行。

我们现在看到的是一个针对`click`事件的通用侦听器。与此类似，每个DOM元素都有自己的`addEventListener`方法，允许侦听特定的元素：

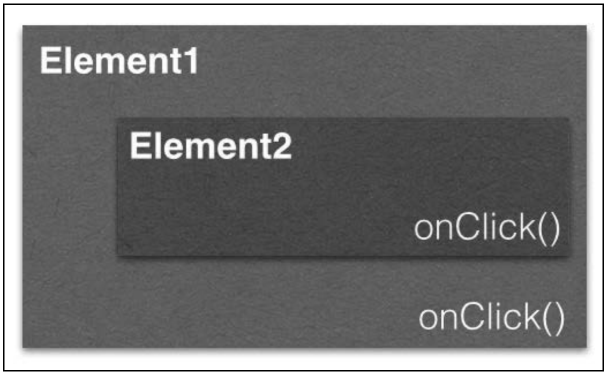
```
<button>Submit</button>
<p>No handler here.</p>
<script>
  var button = document.getElementById("#Bigbutton");
  button.addEventListener("click", function() {
    console.log("Button clicked.");
  });
</script>
```

在本例中，我们通过调用`getElementById()`得到了一个ID为`Bigbutton`的按钮的引用。在该元素的引用上调用`addEventListener()`，为click事件注册了一个事件处理函数。在Mozilla Firefox或Google Chrome等现代浏览器中，这段代码没有任何问题。但是在IE9之前的版本中就不

管用了。原因在于微软没有遵循W3C标准，而是实现了自定义的`attachEvent()`方法。这意味着你不得不依靠一些很麻烦的技巧来处理特定浏览器的怪异行为。

8.5　事件传播

现在我们要问一个重要的问题：如果某个元素及其祖先都注册了同一事件的事件处理程序，哪个事件处理程序会先被触发？考虑下面的示意图：



Element2是Element1的子元素，两者都拥有`onClick`事件处理程序。当用户点击Element2时，Element2和Element1上的`onClick`都会被触发，但问题是谁先谁后？事件发生的顺序是怎样的？遗憾的是，答案完全依赖于浏览器。说到浏览器，自然有两个阵营：网景与微软。

网景认为第一个触发的事件应该是Element1上的`onClick`，这种事件顺序称为事件捕获。

微软认为第一个触发的事件应该是Element2上的`onClick`，这种事件顺序称为事件冒泡。

这是两种完全相反的浏览器事件处理观点及实现。为了结束这种混乱，万维网协会（W3C）采用了一种明智的中间路线。在新的模型中，事件先进入捕获阶段，到达目标元素之后再向上进入冒泡阶段。在这种标准行为中，你可以选择将事件处理程序注册到捕获阶段或是冒泡阶段。如果`addEventListener()`的最后一个参数是true，则事件处理程序注册在捕获阶段；如果是false，则注册在冒泡阶段。

有时候，在事件已经被子元素处理过之后，你并不想让其父元素再经手了。你可以在事件对象上调用`stopPropagation()`方法，阻止事件处理程序进一步接收该事件。有些事件有与之关联的默认行为。例如当你点击了一个URL链接，会被带向链接的目标地址。JavaScript事件处理程序会在默认行为被执行前调用。你可以在事件对象上调用`preventDefault()`方法来阻止执行默认行为。

在浏览器中使用普通的JavaScript时，这些都是事件基础。但这样做存在一个问题。浏览器在

事件处理行为方面的名声实在不怎么样。我们接下来要看看jQuery的事件处理。为了简化处理过程，jQuery总是将事件处理程序注册在冒泡阶段。这意味着最具体的元素能够最先响应事件。

8.6 jQuery 事件处理及传播

jQuery事件处理考虑到了浏览器很多的怪异行为，你可以将注意力放在代码编写上。jQuery对于浏览器事件的支持既简单又直接。例如，下面的代码负责侦听用户点击页面上任何按钮：

```
$('button').click(function(event) {
  console.log('Mouse button clicked');
});
```

与`click()`方法类似，还有其他一些辅助方法，涵盖了几乎所有类型的浏览器事件。这些方法包括：

- `blur`
- `change`
- `click`
- `dblclick`
- `error`
- `focus`
- `keydown`
- `keypress`
- `keyup`
- `load`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `resize`
- `scroll`
- `select`
- `submit`
- `unload`

你也可以使用`.on()`方法，该方法更灵活，它能够为多个事件绑定同一个事件处理程序。利用`on()`方法也可以来处理定制事件。

和其他方法一样，将事件名作为第一个参数传给on()方法：

```
$('button').on( 'click', function( event ) {
  console.log(' Mouse button clicked');
});
```

为元素注册好事件处理程序之后，就可以触发该事件了：

```
$('button').trigger( 'click' );
```

也可以这样触发事件：

```
$('button').click();
```

jQuery的.off()方法可用于解除事件绑定，该方法可以移除绑定到特定事件上的任何事件处理程序：

```
$('button').off( 'click' );
```

你可以向一个元素添加多个事件处理程序：

```
$("#element")
  .on("click", firstHandler)
  .on("click", secondHandler);
```

事件触发时，这两个事件处理程序都会被调用。如果你只想移除第一个事件处理程序，可以使用off()方法，将要移除的那个事件处理程序作为第二个参数：

```
$("#element).off("click",firstHandler);
```

如果你有指向事件处理程序的引用，选择这种做法没有问题。如果事件处理程序采用的是匿名函数，是无法得到对应的引用的。在这种情况下，你可以使用命名空间事件（namespaced event）。考虑下面的例子：

```
$("#element").on("click.firstclick",function() {
  console.log("first click");
});
```

现在你拥有了一个注册在该元素上的命名空间事件处理程序，可以采用如下方式移除：

```
$("#element).off("click.firstclick");
```

采用.on()方法的一个主要优势在于可以一次性绑定多个事件。你可以传入多个以空格分隔的事件名称。考虑下面的例子：

```
$('#inputBoxUserName').on('focus blur', function() {
  console.log( Handling Focus or blur event' );
});
```

你可以为多个事件添加多个事件处理程序：

```
$( "#heading" ).on({
  mouseenter: function() {
    console.log( "mouse entered on heading" );
  },
  mouseleave: function() {
    console.log( "mouse left heading" );
  },
  click: function() {
    console.log( "clicked on heading" );
  }
});
```

在jQuery 1.7中，所有事件都是通过`on()`方法绑定的，即便你调用了如`click()`之类的辅助方法。在内部，jQuery会将这些调用都映射到`on()`方法。正因为如此，一般推荐使用`on()`方法来确保一致性以及更快的执行速度。

8.7 事件委托

事件委托允许我们将单个事件侦听器添加到父元素上。所有匹配选择器的后代元素都能够触发该事件，即使这些后代元素此刻（在绑定侦听器之时）尚未创建。

我们之前讨论过事件冒泡。jQuery中的事件委托主要依靠的就是事件冒泡。只要页面上发生了事件，该事件就会从所源自的元素向上冒泡，直到其父元素，然后再到父元素的父元素，以此类推，直至根元素（`window`）。考虑下面的例子：

```
<html>
  <body>
    <div id="container">
      <ul id="list">
        <li><a href="http://google.com">Google</a></li>
        <li><a href="http://myntra.com">Myntra</a></li>
        <li><a href="http://bing.com">Bing</a></li>
      </ul>
    </div>
  </body>
</html>
```

假设我们希望在点击URL时执行一些公共操作（common action），可以选择为列表中所有的`a`元素添加事件处理程序：

```
$( "#list a" ).on( "click", function( event ) {
  console.log( $( this ).text() );
});
```

这种做法完全没有问题，不过代码中存在一个小bug。如果另一个动态生成的URL被添加到列表中的话，会怎么样呢？假设我们有一个Add按钮，可以用来向列表中添加新的URL。这样一来，如果新的URL作为列表项加入，那么之前的事件处理程序是无法绑定到新的列表项上的。例

8

如，下面的链接被动态地加入到了列表中，点击链接，并不会触发已添加过的事件处理程序：

```
<li><a href="http://yahoo.com">Yahoo</a></li>
```

这是因为事件只会在调用on()方法时注册。在这种情况下，新元素在调用.on()时并不存在，因此也就无法绑定事件处理程序。如果我们理解了事件冒泡，就能够想象得出事件会沿着DOM树向上传播。只要点击了任意一个URL，就出发生以下传播过程：

```
a(click)->li->ul#list->div#container->body->html->root
```

我们可以创建一个委托事件：

```
$( "#list" ).on( "click", "a", function( event ) {
  console.log( $( this ).text() );
});
```

我们将a从最初的选择器变成了on()方法的第二个参数。这个参数告诉事件处理程序监听特定事件，并检查触发该事件的元素是否和第二个参数相同（在这个例子中是a）。如果相同，则执行事件处理程序。有了委托事件，我们就只需要给整个ul#list绑定一个事件处理程序就可以了。这个事件处理程序将侦听由ul元素的后代元素所触发的click事件。

8.8　事件对象

到目前为止，我们都是将匿名函数作为事件处理程序。为了使事件处理程序具有普适性，可以创建命名函数并将其分配给事件。考虑下面的代码：

```
function handlesClicks(event){
  // 处理click事件
}
$("#bigButton").on('click', handlesClicks);
```

这里我们并没有使用匿名函数，而是将一个命名函数传给了on()方法。现在把注意力转到函数的event参数上。jQuery会给所有的事件回调程序传入一个事件对象。事件对象中包含了被触发事件的很多非常有用的信息。如果我们想禁止元素的默认行为，可以使用事件对象的preventDefault()方法。例如，我们想要触发一个AJAX请求，而不是一个完整的表单提交，或者是希望阻止点击URL时跳转到指向的位置。在这些情况下，你可能还想阻止事件在DOM中向上冒泡，这可以通过调用事件对象的stopPropagation()方法来实现。考虑下面的例子：

```
$( "#loginform" ).on( "submit", function( event ) {
  // 阻止表单的默认的提交行为
  event.preventDefault();
  // 阻止事件在DOM树中向上冒泡，停止所有的委托
  event.stopPropagation();
});
```

除了事件对象，你还可以获得触发事件的DOM对象的引用。可以通过`$(this)`来引用该元素。考虑下面的例子：

```
$( "a" ).click(function( event ) {
  var anchor = $( this );
  if ( anchor.attr( "href" ).match( "google" ) ) {
    event.preventDefault();
  }
});
```

8.9 小结

JavaScript最重要的角色就是作为一门浏览器语言，本章全部内容都围绕着这一主题。通过在浏览器中提供DOM操作和事件管理能力，JavaScript为Web页面引入了丰富多彩的变化。我们在使用和不使用jQuery的情况下讨论了这些概念。随着现代化Web的需求日益增长，采用jQuery这类库是非常必要的。这些库能够极大地提高代码质量及效率，同时可以让你放心地把注意力放在重要的方面。

接下来要关注的是JavaScript的另一种实现形式——主要是在服务器端。Node.js已经成为一个流行的JavaScript框架，用于编写可伸缩的服务器端应用程序，我们将详细讲述如何充分利用Node.js来编写服务器端应用程序。

第 9 章

服务器端JavaScript

到目前为止，我们一直都在关注JavaScript作为一门浏览器语言所展现出的多功能性。尤为值得一提的是，JavaScript的卓越表现使其在可伸缩的服务器系统编写方面崭露头角，大受欢迎。在本章中，我们将学习Node.js。Node.js是当前最流行的用于服务器端编程的JavaScript框架。除此之外，它还是GitHub上最受关注的项目，拥有强大的社区支持。

Node使用V8来支持服务器端编程，V8正是Goolge Chrome所使用的虚拟机。V8大大提升了Node的性能，因为它能够将JavaScript直接编译成原生的机器码，这种方式要优于执行字节码或是使用解释器作为中间件。

V8的强大功能加上JavaScript简直是天作之合——JavaScript的性能、功用以及流行程度使得Node一夜成名。在本章中，我们将学习下列主题：

- 浏览器和Node.js的异步事件模型
- 回调函数
- 计时器
- EventEmitter
- 模块和npm

9.1 浏览器的异步事件模型

在学习Node之前，让我们先来理解运行在浏览器中的JavaScript。

Node依赖于事件驱动以及异步平台来实现服务器端的JavaScript。这与浏览器处理JavaScript的方式很相似。浏览器和Node在进行I/O的时候，采用的都是事件驱动及非阻塞的方式。

要想深入理解Node.js的事件驱动及异步特性，首先要比较一下各种操作及其成本：

L1缓冲读操作	0.5纳秒
L2缓冲读操作	7纳秒
内存	100纳秒
4KB SSD随机读操作	150 000纳秒
1MB SSD顺序读操作	1 000 000纳秒
1MB 磁盘顺序读操作	20 000 000纳秒

这些数据取自https://gist.github.com/jboner/2841832，显示出了**输入/输出**（I/O）操作的高昂成本。计算机程序耗时最长的操作就是I/O操作，如果程序总是在等待这些操作完成，则整个程序的执行速度都会被拖慢。让我们来看一个这样的例子：

```
console.log("1");
var log = fileSystemReader.read("./verybigfile.txt");
console.log("2");
```

`fileSystemReader.read()`在调用时会从文件系统中读取文件。正如我们刚刚看到的，这里的瓶颈是I/O，在读取操作结束之前要花费很长的时间完成相关的操作。取决于硬件种类、文件系统、操作系统等因素，整个程序的执行会被这个操作阻塞一段不短的时间。上面的代码执行了一些会阻塞的I/O操作——进程会一直阻塞到I/O结束并有数据返回。这就是大多数人所熟悉的传统I/O模型。这种模型不但成本高，而且会造成可怕的时延。每个进程都有相关的内存和状态——它们都会被阻塞，直到I/O结束。

如果程序阻塞在I/O上，Node服务器会拒绝新的请求。有几种方法可以解决这个问题。最流行的传统方法就是利用多个线程来处理请求，这叫作多线程技术。如果你熟悉Java，那应该写过多线程代码。不同的语言对于线程的支持方式也不同——就本质而言，线程拥有自己的内存和状态。编写大规模的多线程应用并非易事。当多个线程访问一个共享内存或值的时候，在线程间维护正确的状态是一件非常困难的事。在内存和CPU使用率方面，线程的成本也不低。用于同步资源的线程最终也可能会被阻塞。

浏览器对此的处理方式有所不同。浏览器中的I/O发生在主执行线程之外，当I/O结束时会触发一个事件。该事件会由与之关联的回调函数来处理。这种类型的I/O是非阻塞和异步的。因为I/O并不会阻塞主执行线程，所以浏览器可以继续处理出现的其他事件，无需等待任何I/O。这是一种非常有力的概念。异步I/O使得浏览器能够响应多个事件，允许更高层次的交互活动。

Node采用了类似的异步处理方式。Node的事件循环作为一个单独的线程运行。这意味着你所编写的应用实际上是单线程的，但这并不表示Node本身是单线程。Node使用了libuv，是多线程的——好在这些细节都隐藏在Node内部了，在开发应用的时候不需要知道这些。

每一个涉及I/O的调用都需要注册一个回调函数。注册回调函数的过程同样是异步的，立刻就能够返回。I/O操作一结束，对应的回调函数就被推入事件循环中。它会在之前被推入事件循环的所有其他回调函数执行时执行。所有的操作都是线程安全的，这主要是因为在事件循环中没

有需要进行同步的并行执行路径。

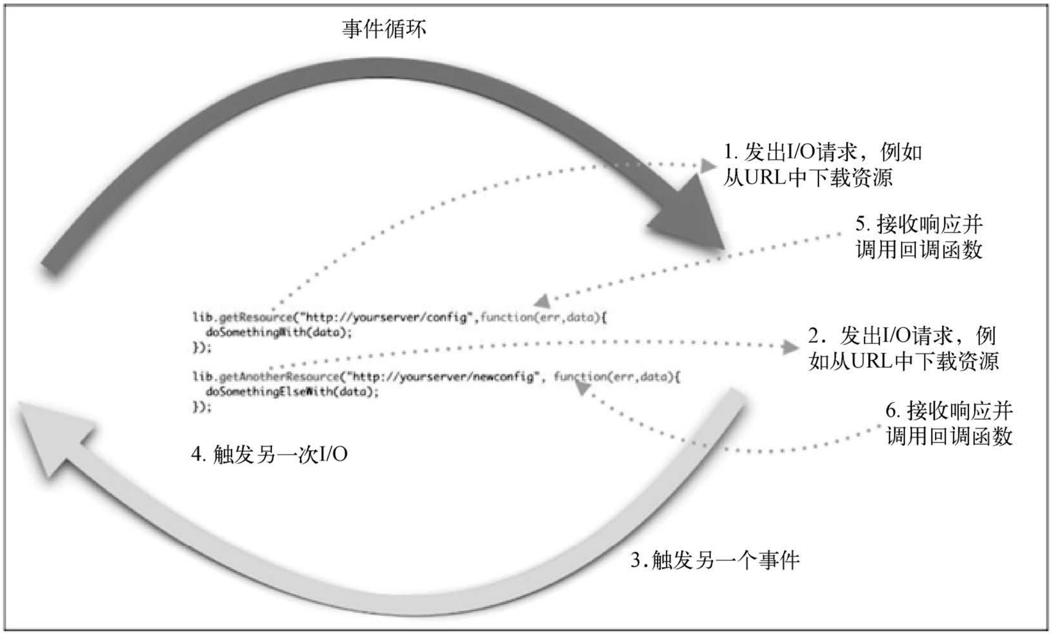
从根本上来说，只有一个线程在运行你的代码，不存在并行执行；但除此之外的都是并行的。

Node.js依赖于libev（http://software.schmorp.de/pkg/libev.html）提供事件循环，由libeio（http://software.schmorp.de/pkg/libeio.html）使用线程池来提供异步I/O支持。要了解更多的相关内容，可以查阅libev的文档：http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod。

下面是一个Node.js中异步代码执行的例子：

```
var fs = require('fs');
console.log('1');
fs.readFile('./response.json', function (error, data) {
  if(!error){
    console.log(data);
  });
console.log('2');
```

在这个程序中，我们从磁盘读取`response.json`文件。当磁盘I/O结束后会执行指定的回调函数，在函数参数中包含了可能出现的错误以及读取到的文件数据。你会在控制台中看到`console.log('2')`的输出内容紧随着`console.log('1')`的输出:



Node.js不需要任何额外的服务器组件，因为它会创建自己的服务器进程。一个Node应用实际上就是在指派接口上运行的一个服务器。在Node中，服务器和应用是相同的。

这里有一个Node.js服务器的例子，当在浏览器中输入http://localhost:3000/时，它会使用字符串`Hello Node`作为响应：

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello Node\n');
});
server.listen(3000);
```

在本例中，我们使用了`http`模块。回忆一下我们之前关于JavaScript模块的讨论，你会发现这正是CommonJS模块的实现。Node将多个模块编译成二进制形式。核心模块定义在Node的源代码中。这些模块位于lib/目录中。

如果模块名被传入`require()`，会首先载入指定的模块。例如，`require('http')`会载入内置的HTTP模块，即使有其他文件也使用这个名字。

载入用于处理HTTP请求的模块之后，我们创建了一个`server`对象，并使用`server.on()`方法来侦听`request`事件。只要有请求到达端口`3000`，就会调用相应的回调函数。回调函数使用`request`和`response`作为参数。另外，在回送响应信息之前，我们设置了`Content-type`头部和HTTP响应代码。你可以把上面的代码复制下来，保存成纯文本文件并命名为app.js，然后使用Node.js在命令行中运行这个服务器：

```
$ >> node app.js
```

服务器启动之后，可以在浏览器的地址栏中输入http://localhost:3000，然后就可以看到下面这行平淡无奇的文本了：



如果想查看内部细节，可以使用`curl`命令：

```
~ » curl -v http://localhost:3000
* Rebuilt URL to: http://localhost:3000/
*   Trying ::1…
* Connected to localhost (::1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
```

```
< Content-Type: text/plain
< Date: Thu, 12 Nov 2015 05:31:44 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
 Hello Node
* Connection #0 to host localhost left intact
```

curl会使用表示请求（>）和响应（<）的指示符显示出对应的请求头部和响应头部信息。

9.2　回调函数

JavaScript中的回调函数不容易上手，通常需要一些时间来适应。如果你之前从事的是非异步编程，那需要仔细地学习并理解回调函数的工作原理，你可能会发现自己好像第一次学编程一样。由于Node中的一切都是异步的，所以你将使用回调函数，而不需要仔细地构造它们。Node.js项目中最重要的部分是代码组织与模块管理。

回调函数是随后以异步形式执行的函数。异步程序并不是从上到下顺序执行，它会根据某些事件（如HTTP请求或文件系统读取操作）出现的顺序和速度，在不同的时间执行不同的函数。

一个函数究竟是顺序执行还是异步执行，依赖于它执行时所处的上下文：

```
var i=0;
function add(num){
  console.log(i);
  i=i+num;
}
add(100);
console.log(i);
```

如果使用Node运行这个程序，你会看到下列输出（假设文件名是app.js）：

```
~/Chapter9 » node app.js
0
100
```

这是我们习惯看到的结果。传统的同步代码在执行时是逐行顺序执行的。上面的代码中定义了一个函数，然后接着在下一行调用该函数，不做任何等待。这就是顺序控制流程。

要是在顺序执行序列中引入了I/O，情况会有所不同。如果我们尝试从文件中读取内容或者调用远程端点（remote endpoint），Node会采用异步方式执行这些操作。下例中，我们要使用一个叫作request的Node.js模块。该模块用来发起HTTP调用，安装方法如下：

```
npm install request
```

我们在本章随后部分将讨论npm的用法。考虑下面的例子：

```
var request = require('request');
var status = undefined;
request('http://google.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    status_code = response.statusCode;
  }
});
console.log(status);
```

在执行这段代码时，你会发现变量status的值依然是undefined。在本例中，我们发起了一个HTTP调用——这属于I/O操作。当进行I/O操作时，执行过程就变成异步了。在前面的例子中，所有的操作都在内存中，不涉及I/O，因此执行过程是同步的。当我们运行这个程序时，所有的函数立刻被定义，但并不是马上就执行。调用request()函数后，继续执行下一行代码。如果没有可执行的内容，Node要么等待I/O结束，要么退出。当request()函数完成其任务之后，会接着执行回调函数（作为request()函数第二个参数的那个匿名函数）。上面的例子中出现undefined的原因在于，代码中没有对应的逻辑告诉console.log()语句应该等到request()函数结束之后再从HTTP调用中获取响应信息。

回调函数并不会立刻执行，这改变了代码的组织方式。可以按照下面的方式重新组织代码：

- 将异步代码放入函数
- 将回调函数传给包裹函数（wrapper function）

我们遵照上面两个要点重新组织之前的例子。来看修改后的代码：

```
var request = require('request');
var status = undefined;
function getSiteStatus(callback){
  request('http://google.com', function (error, response, body) {
    if (!error && response.statusCode == 200) {
      status_code = response.statusCode;
    }
    callback(status_code);
  });
}
function showStatusCode(status){
  console.log(status);
}
getSiteStatus(showStatusCode);
```

运行后会得到如下（正确的）输出：

```
$node app.js
200
```

我们将异步代码封装到了函数getSiteStatus()中，把名为callback()的函数作为参数传入，并在getSiteStatus()的最后一行执行该函数。回调函数showStatusCode()只是简单地包裹了之前调用的console.log()。然而，不同之处在于异步执行的方式。在学习如何使用

回调函数编程时，最重要的一点就是要明白函数是头等对象（first-class object），可以保存在变量中，也可以作为参数传递。为变量起一个简单且富有描述性的名字对于代码的可读性非常重要。现在，一旦HTTP调用完成，回调函数就会被调用，变量status_code中就会得到正确的值。在有些实际情况中，你希望某个异步任务仅在其他异步任务完成之后才执行。考虑下面的情形：

```
http.createServer(function (req, res) {
  getURL(url, function (err, res) {
    getURLContent(res.data, function(err,res) {
    ...
    });
  });
});
```

可以看到，我们将一个异步函数嵌套到了另一个异步函数中。这种嵌套形式会导致代码难以阅读和管理。这样的回调方式有时被称为**回调函数地狱**（callback hell）。为了避免这种情况，如果你有一些代码必须等待其他异步代码结束，可以通过将代码放入函数，再将其作为回调函数传递，以此传达出相互之间的依赖性。另一个要点是为函数起个名字，而不是依赖使用匿名函数作为回调函数。我们可以对前面的示例进行重构，使其更有可读性，如下所示:：

```
var urlContentProcessor = function(data){
  ...
}
var urlResponseProcessor = function(data){
  getURLContent(data,urlContentProcessor);
}
var createServer = function(req,res){
  getURL(url,urlResponseProcessor);
};
http.createServer(createServer);
```

这段代码中体现了两个重要的概念。首先，我们使用了命名函数并将其作为回调函数。其次，没有出现异步函数嵌套。如果在内部函数中需要访问闭包变量，则需要略作修改。在这个例子中，行内匿名函数是一种更好的选择。

在Node中会经常用到回调函数。它们更适合于定义那些一次性响应的处理逻辑。如果需要响应重复性事件，Node还提供了其他机制。在进一步深入之前，我们需要理解Node中的计时器功能以及事件。

9.3　计时器

计时器用于在一定延迟之后调度执行特定的回调函数。设置这种延迟执行的主要方法有两种：setTimeout和setInterval。setTimeout()函数用于延迟之后调度执行回调函数，而setInterval用来重复地调度执行回调函数。setTimeout()函数对于需要定时执行的任务非常有用，比如说整理工作。考虑下面的例子：

```
setTimeout(function() {
  console.log("This is just one time delay");
},1000);
var count = 0;
var t = setInterval(function() {
  count++;
  console.log(count);
  if (count> 5){
    clearInteval(t);
  }
}, 2000 );
```

首先，我们使用`setTimeout()`安排在1000毫秒之后执行一个回调函数（匿名函数）。这个回调函数只调度执行一次。我们使用`setInterval()`来安排重复执行另一个回调函数。注意，`setInterval()`的返回值被赋给了变量`t`——可以在`clearInterval()`中使用该引用来清除此次调度活动。

9.4 EventEmitter

我们之前讨论过，回调函数非常适合执行一次性逻辑处理（one-off logic）。EventEmitter在响应重复事件方面非常有用。EventEmitter可以触发事件，也能够处理这些被触发的事件。有些重要的Node API就是基于EventEmitter构建的。

由EventEmitter生成的事件是通过侦听器处理的。侦听器是一个与事件相关联的回调函数——当事件发生时，相关的侦听器也会被触发。`event.EventEmitter`是一个类，用于为发射（触发）事件和绑定回调函数提供一致的接口。

按照惯例，事件名采用驼峰命名法；不过，任何有效的字符串都可以用作事件名称。

使用`require('events')`来访问`EventEmitter`类：

```
var EventEmitter = require('events');
```

如果EventEmitter实例发生错误，会引发`error`事件。在Node.js中，error事件被视为一种特殊情况。如果不进行处理的话，程序会携带异常栈（exception stack）退出。

如果添加或删除了侦听器，EventEmitter会分别引发出`newListener`事件或`removeListener`事件。

为了理解EventEmitter的用法，我们编写了一个简单的telnet服务器，不同的客户端都可以登录并输入一些命令。服务器会根据不同的命令作出相应的响应：

```
var _net = require('net');
var _events = require ('events');
var _emitter = new events.EventEmitter();
```

```
_emitter.on('join', function(id,caller){
  console.log(id+" - joined");
});
_emitter.on('quit', function(id,caller){
  console.log(id+" - left");
});
var _server = _net.createServer(function(caller) {
  var process_id = caller.remoteAddress + ':' + caller.remotePort;
  _emitter.emit('join',id,caller);
  caller.on('end', function() {
    console.log("disconnected");
    _emitter.emit('quit',id,caller);
  });
});
_server.listen(8124);
```

这段代码中使用了Node的`net`模块。我们想做的是创建一个服务器，让客户端可以通过标准的`telnet`命令进行连接。当客户端成功连接后，服务器会显示出客户端的地址和端口号；当客户端退出时，服务器会将同样的信息记录到日志中。

客户端连接时会引发`join`事件，在断开时会引发`quit`事件。我们为这两个事件都设置了侦听器，这些侦听器会在服务器端记录下相应的信息。

启动这个程序，使用telent连接到服务器：

```
telnet 127.0.0.1 8124
```

在服务器控制台中，你会看到服务器已经记录下已连接的客户端：

```
» node app.js
::ffff:127.0.0.1:51000 - joined
::ffff:127.0.0.1:51001 - joined
```

如果有客户端退出会话，也会出现相应的信息。

9.5　模块

当你编写了大量代码之后，接下来要考虑的就是如何组织代码。Node采用的是我们之前在讲模块模式时讨论过的CommonJS模块。Node模块可以发布到**Node包管理器**（Node Package Manager，npm）仓库中。npm仓库是一个在线的Node模块合集。

创建模块

Node模块可以是单个文件，也可以是包含一个或多个文件的目录。通常最好是创建一个单独的模块目录。模块目录中的文件一般都命名为index.js。模块目录的结构如下所示：

```
node_project/src/nav
                --- >index.js
```

在项目目录中，模块目录nav包含了模块代码。通常，模块代码需要放在index.js文件中——你也可以将其修改成其他文件。考虑下面这个名为geo.js的简单模块：

```
exports.area = function (r) {
  return 3.14 * r * r;
};
exports.circumference = function (r) {
  return 3.14 * 3.14 * r;
};
```

你可以通过exports导出这两个函数。使用这个模块时需要用到require()函数，这个函数接受模块名或是指向模块代码的系统路径。可以像下面这样使用该模块：

```
var geo = require('./geo.js');
console.log(geo.area(2));
```

因为我们只向外部导出了两个函数，所以模块中其余的部分仍旧是私有状态。回想一下曾经讲解过的模块模式——Node使用的就是CommonJS模块。还有另外一种创建模块的语法。你可以使用modules.exports来导出模块。实际上，exports只是modules.exports的一个辅助工具。在使用exports时，它会将模块导出的属性赋给modules.exports。但如果modules.exports已经有了这些属性，就会将其忽略。

为了返回一个Geo构造函数，而非包含函数的对象，需要重写先前创建的geo模块。重写后的geo模块及其用法如下：

```
var Geo = function(PI) {
  this.PI = PI;
}
Geo.prototype.area = function (r) {
  return this.PI * r * r;
};
Geo.prototype.circumference = function (r) {
  return this.PI * this.PI * r;
};
module.exports = Geo;
```

考虑一个config.js模块：

```
var db_config = {
  server: "0.0.0.0",
  port: "3306",
  user: "mysql",
  password: "mysql"
};
module.exports = db_config;
```

如果你想从模块外部访问db_config，可以使用require()来包含该模块，然后像下面这

样引用对象：

```
var config = require('./config.js');
console.log(config.user);
```

组织模块的方式有三种。

- 使用相对路径，例如：`config = require('./lib/config.js')`
- 使用绝对路径，例如：`config = require('/nodeproject/lib/config.js')`
- 搜索模块，例如：`config = require('config')`

前两种方式不言而喻——让Node在文件系统的特定位置查找模块。

如果你选择第三种方式，也就是要求Node使用标准查找方法来定位模块。为了定位到指定的模块，Node从当前目录开始，在其后加上./node_modules/，然后尝试从这个位置载入模块。如果模块没有找到，那就从父目录开始再查找，直至文件系统的根目录。

举例来说，如果在/projects/node/中调用`require('config')`，会搜索以下位置，直至找到匹配项为止：

- /projects/node /node_modules/config.js
- /projects/node_modules/config.js
- /node_modules/config.js

对于从npm下载的模块，使用这种方法要相对简单。正如之前讲过的，只要为Node提供一个入口点，就可以在目录中组织模块了。

最简单的方法就是创建一个./node_modules/supermodule/目录，在其中放入index.js文件。该文件在默认情况下会被载入。还有一种方法，你可以在mymodulename目录下放置一个package.json文件，在其中指定模块名及其主文件名：

```
{
  "name": "supermodule",
  "main": "./lib/config.js"
}
```

你得知道的是Node会像缓存对象那样缓存模块。如果你有两个（或多个）文件需要某个模块，第一个`require`会将该模块缓存到内存中，这样第二个`require`就不必重新载入模块源代码了。但如果需要的话，后一个`require`可以修改模块的功能。这通常叫作**猴子补丁**（monkey patching），可以在不需要修改原始模块的情况下改变模块的行为。

9.6　npm

npm是Node用于发布模块的包管理器。npm可以用于安装、更新及管理模块。包管理器在Python等其他语言中也很流行。npm能够自动解决和更新包的依赖关系，从而减轻你的负担。

安装包

安装npm包的方法有两种：局部安装和全局安装。如果只想在特定的Node项目中使用某个模块的功能，你可以针对该项目进行局部安装，这是`npm install`的默认行为。另外，有一些模块可以用作命令行工具；在这种情况下，可以选择全局安装：

```
npm install request
```

`npm`的`install`指令能够安装特定的模块——在本例中是`request`模块。要确认`npm install`是否执行顺利，可以查看node_modules目录是否存在，并检查是否包含所安装包的目录。

当把模块添加到项目中之后，管理每个模块的版本/依赖性就变得非常困难。对于以局部形式安装的包，最佳管理方法是在项目中创建package.json文件。

package.json文件有下列好处。

- 定义待安装的每个模块的版本。有时候，你的项目要依赖于某个模块的特定版本。在这种情况下，package.json能够帮助你下载并维护正确的版本依赖关系。
- 作为项目中所用到的全部模块的文档。
- 在部署和打包应用程序时无需担心部署代码时的依赖管理。

可以使用以下命令创建package.json文件：

```
npm init
```

在回答有关项目的基本问题之后，一个空白的package.json文件就创建好了，内容如下：

```
{
  "name": "chapter9",
  "version": "1.0.0",
  "description": "chapter9 sample project",
  "main": "app.js",
  "dependencies": {
    "request": "^2.65.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Chapter9",
    "sample",
```

```
      "project"
    ],
    "author": "Ved Antani",
    "license": "MIT"
  }
```

你可以在文本编辑器中手动编辑这个文件。该文件中一个重要的部分就是`dependencies`标签。要想指定项目所依赖的包，需要将其在package.json文件中列出。这些包的类型有两种。

- `dependencies`：在生产环境中所需要的包。
- `devDependencies`：仅用于开发和测试环境的包（例如使用Jasmine node package）。

在前面的例子中，你可以看到以下依赖关系：

```
"dependencies": {
  "request": "^2.65.0"
},
```

这表示该项目依赖`requset`模块。

> 模块的版本依赖于语义化的版本规则：https://docs.npmjs.com/getting-started/semantic-versioning。

package.json文件准备好之后，就可以使用`npm install`命令为项目自动安装模块了。

有一个很酷的技巧，我个人爱不释手。在从命令行安装模块时，可以使用`--save`选项将该模块的依赖关系自动添加到package.json文件中：

```
npm install async --save
npm WARN package.json chapter9@1.0.0 No repository field.
npm WARN package.json chapter9@1.0.0 No README data
async@1.5.0 node_modules/async
```

在上面的命令中，我们使用带有`--save`选项的`npm`命令安装了`async`模块。在package.json文件中会自动创建相应的条目：

```
"dependencies": {
  "async": "^1.5.0",
  "request": "^2.65.0"
},
```

9.7　JavaScript 性能

与其他语言一样，编写大规模JavaScript代码时保证正确性是必不可少的。随着这门语言的日益成熟，有些先天的问题需要留意。一些表现不凡的库在编写高质量代码方面做出了贡献。对于大多数重大系统而言，**良好的代码 = 正确的代码 + 高性能的代码**。新一代软件系统非常强调性能。在本节中，我们将讨论一些可以用来分析JavaScript代码并理解其性能标准的工具。

我们接下来会讲到两个概念。

- 性能分析（profiling）：在脚本分析过程中对各种函数和操作进行计时，有助于鉴别出代码中可优化的部分。
- 网络性能：检查如图片、样式表单、脚本等网络资源的载入。

JavaScript 性能分析

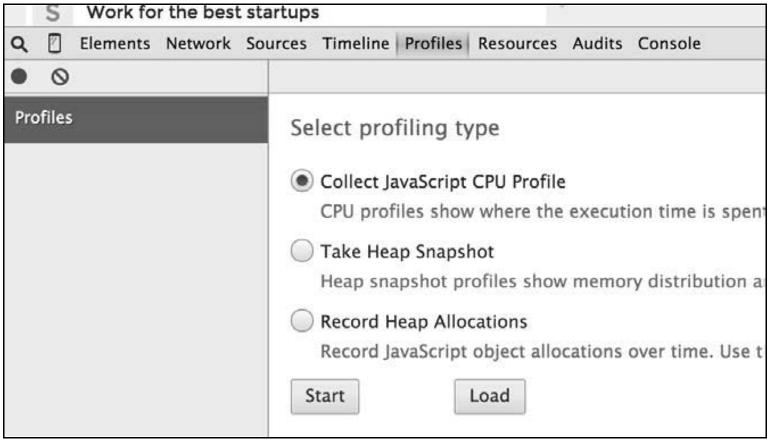
JavaScript性能分析对于理解代码中不同部分的性能非常重要。你可以观察函数和操作的用时情况，了解哪些操作花费的时间较长。有了这些信息，你就可以优化那些耗时函数的性能，调校代码的整体性能。我们将把注意力放在Chrome的Developer Tools提供的性能分析选项上，其中有各式各样的分析工具可供你理解代码的性能标准。

1. CPU分析

CPU分析会显示出代码中各部分所耗费的执行时间。我们必须提醒DevTools记录CPU分析数据。来上手一试吧。

在DevTools中启用CPU分析器的方法如下。

(1) 打开Chrome DevTools的Profiles面板。
(2) 检查Collect JavaScript CPU Profile是否选中：

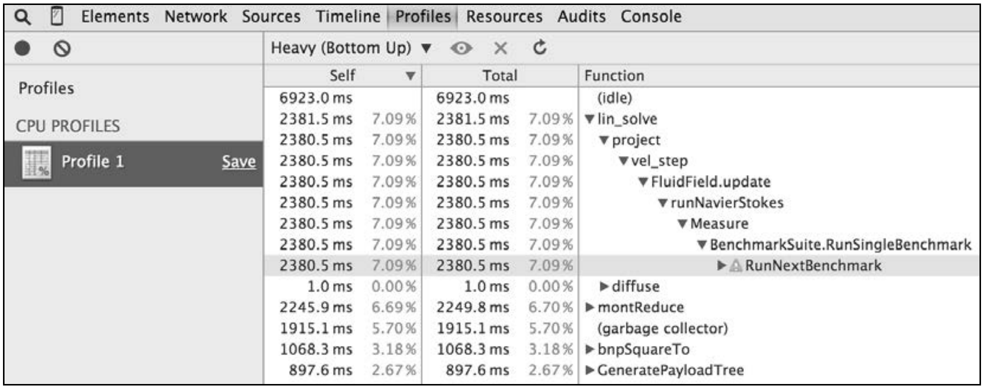


在本章中，我们将使用Google的基准测试（benchmark）页面：http://octanebenchmark.googlecode.com/svn/latest/index.html[①]。我们使用它，是因为它包含了一些示例功能，我们可以从

① 在翻译本书之时，该页面已无法访问。具体的页面地址，读者可自行以V8 Benchmark Suite为关键字搜索。——译者注

中观察到各种性能瓶颈和基准。要开始记录CPU分析，需要在Chrome中打开DevTools，在Profiles标签下单击Start按钮或按Cmd/Ctrl+E。刷新V8 Benchmark Suite页面。当该页面完全重新加载之后，将显示基准测试成绩。返回到Profiles面板，单击Stop按钮停止记录，或是再次按Cmd/Ctrl+E。

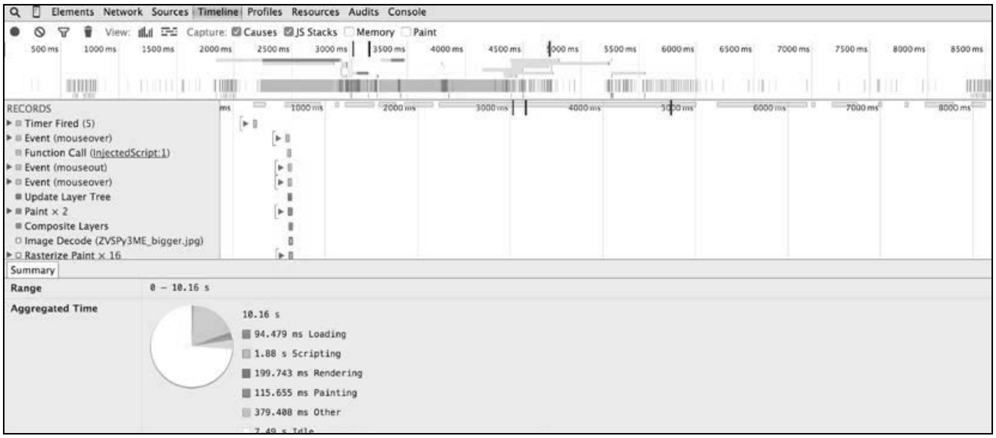
已记录下的CPU分析数据以自下而上的方式显示出了各项功能的执行时间，如下图所示：



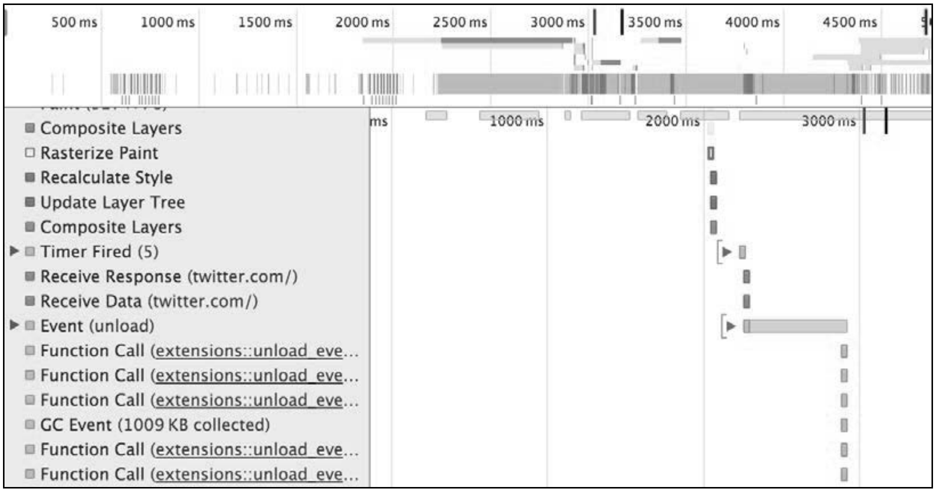
2. Timeline视图

Chrome DevTools的Timeline（时间线）工具是查看代码整体性能时第一个要去的地方。它能够记录和分析应用程序在运行时的所有活动。

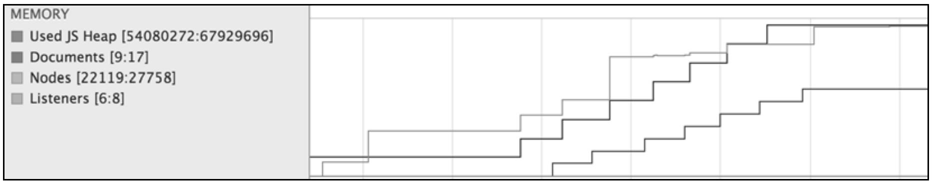
Timeline能够让你全面了解在载入和使用网站的过程中，时间都耗费在了哪些地方。每一个发生的事件都被记录在内，并以**瀑布图**的形式展现出来：



上面的截图展示了在浏览器中渲染https://twitter.com/时的时间线视图。利用该视图，你就可以全面掌握哪些操作耗费了多少执行时间：



在这个截图中，我们可以看到各种JavaScript函数、网络调用、资源下载以及其他Twitter主页渲染操作的渐进式执行（progressive execution）。通过该视图，我们能够充分了解哪些操作耗时更长。只要找出了这些操作，就可以对其进行性能优化。Memory视图是一个挺不错的工具，可以用来理解浏览器中的应用在其执行周期内的内存占用情况。在Memory视图中显示了应用程序在一段时间内的内存占用图，同时还维护了一个能够统计内存中所含文档、DOM节点及事件侦听器数量的计数器。除此之外，该视图还有助于检测内存泄漏并给出清晰的提示，告诉你需要进行哪些优化：



JavaScript性能是一个令人着迷的话题，值得专门来讲述。强烈建议你好好研究Chrome的DevTools，弄清楚如何充分利用该工具检测和诊断代码中的性能问题。

9.8　小结

在本章中，我们看到了JavaScript的另一个化身——以Node.js形式存在的服务器端框架。

Node提供了一个异步事件模型，用于在JavaScript中编写可伸缩和高性能的服务器应用程序。我们深入学习了Node的一些核心概念，例如事件循环、回调函数、模块和计时器。理解这些概念对于编写良好的Node代码至关重要。我们还讨论了一些用于更好地建构Node代码及回调函数的技术。

至此，这门杰出的编程语言的探索之旅也该作结了。JavaScript的无所不能使其已经成为万维网演变的推动力量。它仍在继续扩展着自己的疆土，在每次新的迭代中不断改进。

一开始我们了解了JavaScript语法的基本组成部分，掌握了闭包的基本概念以及函数的用法。大部分JavaScript模式都基于这些基本概念。我们研究了如何利用这些模式编写出更好的JavaScript代码，另外还学习了JavaScript如何处理DOM以及如何使用jQuery有效地操作DOM。最后，我们见识了JavaScript在服务器端的化身：Node.js。

本书能够为你在使用JavaScript编程时开启另一种思考方式。你不仅能够考虑到常见的模式，还能理解并使用ES6所带来的新语言特性。

延 展 阅 读



- 打通JavaScript语言的任督二脉
- 领略语言内部的绝美风光

书号：978-7-115-38573-4
定价：49.00 元



- 深入挖掘JavaScript语言本质
- 简练形象地解释抽象概念

书号：978-7-115-43116-5
定价：79.00 元



- Head First系列经典畅销书
- 囊括JavaScript语言的方方面面
- 为独立打造Web应用程序打下坚实的基础

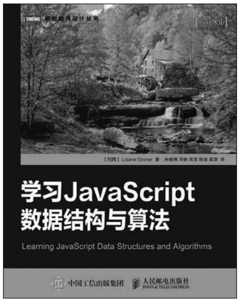
书号：978-7-115-45841-4
定价：129.00 元



- jQuery之父经典著作新版
- 系统总结JavaScript语言特点，直击JavaScript本质

书号：978-7-115-43286-5
定价：45.00 元

延 展 阅 读



用JavaScript学习最常用的数据结构和算法，高效解决计算机科学中的常见问题

第 2 版即将出版
书号：978-7-115-40414-5
定价：39.00 元



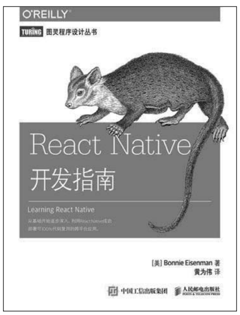
- 构建先行
- 设计干净、可测试、结构良好的JavaScript应用

书号：978-7-115-40210-3
定价：59.00 元



- Red Hat公司真实案例分析
- 教你掌握前端架构设计四个核心，让前端开发可持续优化、可扩展

书号：978-7-115-45236-8
定价：49.00 元



- 从基础开始逐步深入
- 利用React Native成功部署可100%代码复用的跨平台应用

书号：978-7-115-42526-3
定价：59.00 元

JavaScript是一门高级、动态、无类型、轻量的解释型编程语言，是创造万维网内容必不可少的技术之一。大部分网站都使用了JavaScript，而且所有的现代浏览器不需要插件就可以很好地支持它。但是近几年来，JavaScript的方方面面发生了巨大的变化，而你需要适应这个JavaScript新世界。

本书是全面掌握JavaScript、构建创新型Web应用的良好伴侣。书中介绍了JavaScript的语言构件、变量作用域、循环、类型和数据结构的最佳使用实践、JavaScript代码风格，以及推荐的代码组织模式等。如果你是Web开发者，想要掌握现代JavaScript概念和设计原则，本书便是你的理想之选。

- 了解基本的JavaScript语言构件
- 熟悉JavaScript的函数与闭包
- 探究JavaScript的正则表达式
- 利用JavaScript强大的面向对象特性编程
- 使用JavaScript的策略测试及调试代码
- 掌握DOM操作、跨浏览器策略以及ES6
- 理解基本的JavaScript并发构件以及最佳的性能策略
- 学习使用Node.js构建可伸缩的服务器应用

[PACKT]
PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/前端开发/JavaScript

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46569-6

ISBN 978-7-115-46569-6
定价：39.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks